# Introduction to the FHP Architecture

DOCUMENT NUMBER: 069_600002_00.20

AUTHORS: Tom Ash

DATE: July 30, 1980

ABSTRACT: This is revision 5 of the Introduction to the FHP Architecture. The draft contains production illustrations and lineprint captions. The draft has not been signed-off for technical content. It has been released for technical review, but the reviewer's comments have not been incorporated into the text.

KEYWORDS: None

F-DRAWINGS ACCOMPANYING THIS DOCUMENT: None

Contents

## Figures

Tables

# Preface

Introduction to the FHP Architecture is an overview of Data General's FHP architecture. This new architecture supports high-performance systems with long, cost-effective life spans. The FHP architecture is designed to efficiently support a wide range of high-level languages.

This manual describes many user-visible and user-invisible features of the FHP architecture. These features make FHP's design superior to conventional architectures.

Introduction to the FHP Architecture does not contain rigid or complete architectural statements. We describe non-architectural FHP implementation techniques to show the architecture's inherent flexibility.

This manual is intended for use by:

* prospective FHP system customers; and
* Data General FHP, ECLIPSE, NOVA users.

This manual is written for persons with a bachelor's degree in a technical discipline, or a minimum of two years programming or system experience.

Introduction to the FHP Architecture has three chapters and an appendix. Chapter 1 introduces the FHP architecture and briefly describes why it was developed. Chapter 2 describes specific FHP features. Chapter 3 shows how these features are integrated into FHP's high-level architecture and how the architecture can be implemented. We define new terms as they are introduced; the Appendix is a glossary of FHP terms.

Notes

* This manual uses the convention that a byte is 8 bits and that a K byte is equal to 2**10 (1024) bytes. An M byte and a megabyte are equal to 2**20 (1,048,576) bytes.
* To order any Data General manual, contact your sales representative and supply the manual title and ordering number.
* We use third-person masculine pronouns in this manual in a purely generic sense to avoid awkward grammatical

constructions.

* We would appreciate your comments on this manual. Please take the time to record your reactions and suggestions on the form provided on the last page.

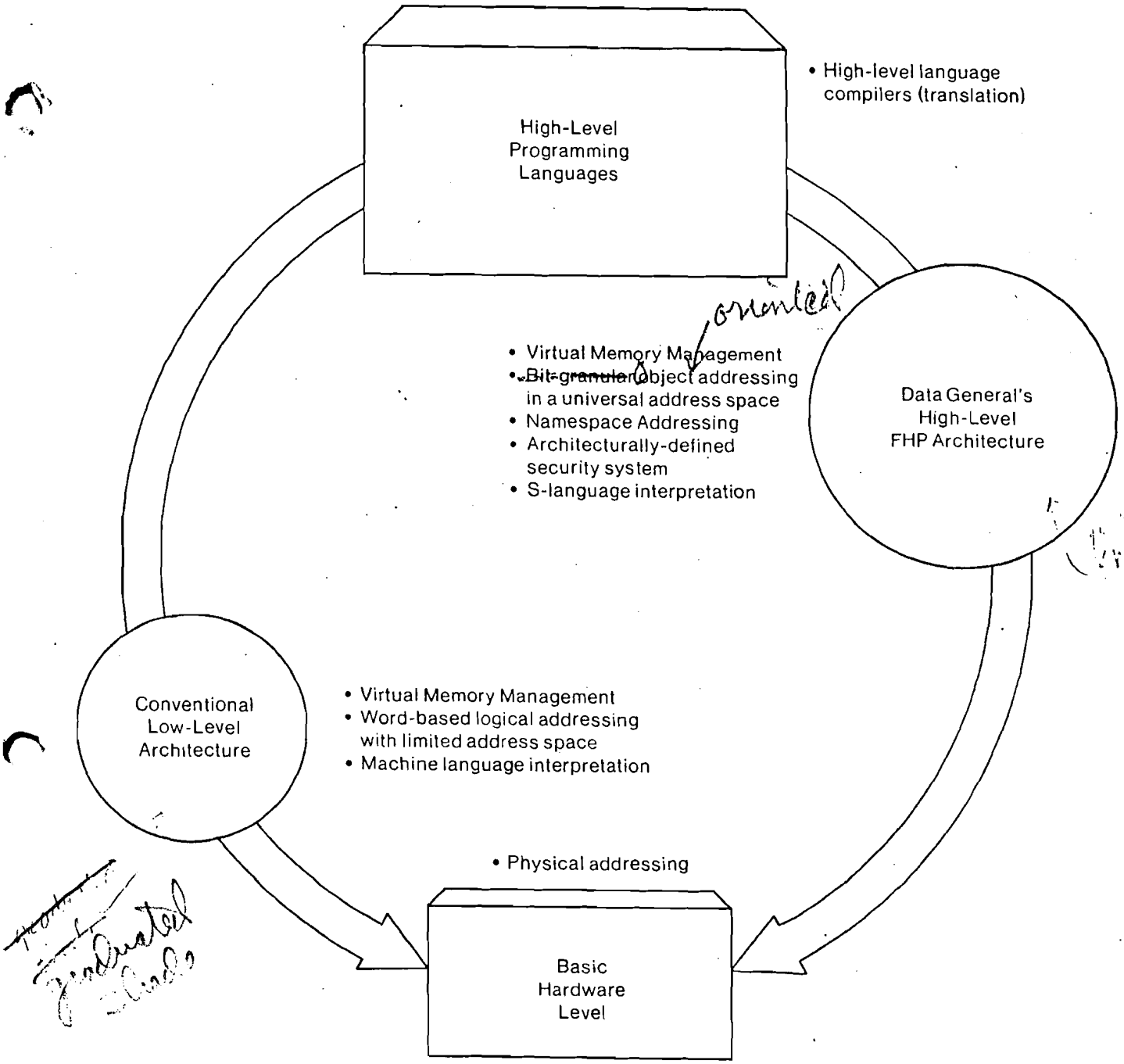
Prerequisite Manuals

* EHP_SPRINI_Product_Summary (014-005000)

Other Related Manuals

* EHP_SPRINI_Theory_of_Operation (014-005002)

--End of Preface--




CAPTION: Data General's new FHP architecture provides many high-level features that conventional architectures lack. The FHP architecture is designed to take advantage of advances in hardware and software technology. Features such as object-oriented addressing let FHP systems run high-level language programs more effeciently than conventional systems do.

# Chapter 1
# Introduction to the Architecture

This manual introduces the major features of Data General's FHP architecture. This new architecture efficiently supports a wide range of high-level languages. The architecture was developed by Data General's Advanced Systems Group in Research Triangle Park, North Carolina.

FHP systems essentially eliminate programmers' concerns about their system's high-level language efficiency, memory management, and program and data security. FHP systems can use different hardware and firmware without changing a high-level language programmer's perception of the system. The architecture supports the implementation of multi-programming and multi-processing systems, which can service multiple users efficiently. The FHP architecture has many features that are invisible to high-level language programmers. However, we describe these features to demonstrate the advantages of the FHP architecture over conventional architectures. We recommend that you read the FHP SPRINT Product Summary (014-005000) before reading this manual, because it introduces many features described in this manual.

In this chapter, we explain why we developed this powerful new architecture and what our design goals were; we then introduce the FHP architecture's major features. These features are discussed in detail in the following chapters.

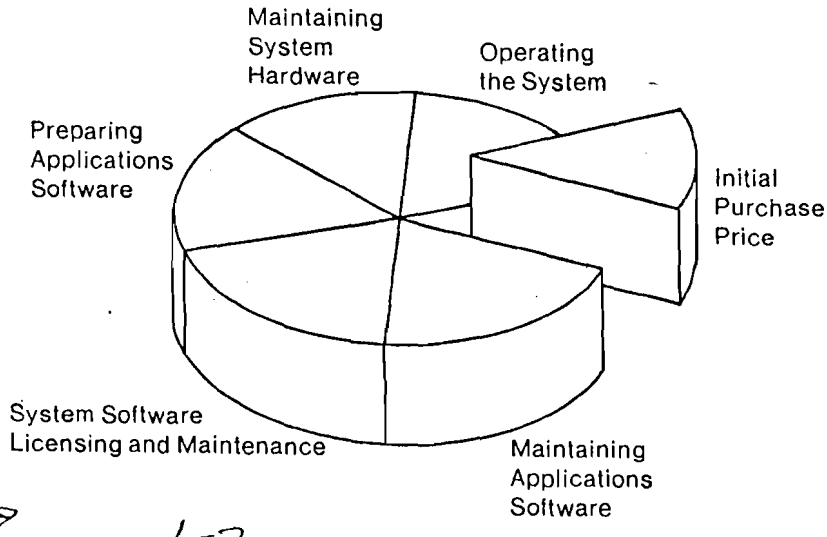
tc22

Figure 1-1. FHP's High-Level Architecture

## 1.1 Design Considerations

Why has Data General developed a new architecture, and what considerations influenced its design? To answer this question, we should first review the reason you use a computer system.

You use a computer system as a tool to increase productivity or reduce operating costs. Like all operating assets, a system must meet two important criteria: it must respond to your short-term and long-term requirements, and must justify its cost over its projected lifetime. You justify a system's cost by comparing its performance to other methods of performing the same tasks.




TC-00023-00 1-2

CAPTION: A computer system's initial purchase price is generally a small part of its long-term cost. The FHP architecture helps reduce other system costs, especially the cost of preparing and maintaining software.

Six major factors contribute to the total cost of a computer system:

1) The initial purchase price of the system

2) Operating the system

3) Maintaining the system hardware

4) Preparing applications software

5) System software licensing and maintenance

6) Maintaining applications software

The cost of each factor differs for every system application. In large applications, the initial purchase price is usually a small part of the total system cost. The long-term costs of operating and maintaining system hardware and software can far exceed initial costs.

tc23

Figure 1-2. Computer System Costs

For most large or diverse applications, the most significant system cost is the cost of preparing, processing, and maintaining application software. Therefore, Data General specifically designed the FHP architecture to reduce software related costs, as well as the cost of the associated hardware.

When you use or supply computing services, your productivity is measured in terms of programming and CPU efficiency, and you try to balance efficiency against system cost. Studies have shown that programmers can produce only a limited number of lines of debugged and documented code in a given period of time. High-level languages have almost completely replaced assembly languages, because they produce more processing work for each instruction and are easier to prepare and maintain. Therefore, FHP systems are designed to be programmed exclusively in high-level languages.

After investigating conventional architectures, we found that our goal of reducing software costs required a combination of features not available in any one conventional architecture. Some of the costs of preparing, processing, and maintaining software can be attributed to restrictions of conventional architectures. These restrictions include a single machine language, a limited address

space, and word-based addressing. In fact, conventional architectures have not kept pace with advances in hardware and software technology. Our goal of reducing software costs required that we develop a new architecture.

## 1.2 FHP Design Goals and Major Features

FHP is a balanced architecture that supports a broad range of systems with long, cost-effective life spans. The following design goals were used to develop FHP's architecture.

* High-level language efficiency
* Addressing flexibility
* Security
* Technological growth potential

The next four sections in this chapter introduce the features of the FHP architecture that meet these design goals.

### High-level Language Efficiency

A flexible architecture must efficiently process a wide range of high-level languages so that programmers can use application-efficient, rather than machine-efficient, languages. FHP systems use a set of machine-language substitutes, called S-languages. FHP's S-language instructions contain more information than conventional assembly language instructions do. Each S-language is closely matched to the needs of a high-level language such as COBOL or FORTRAN.

FHP is a language-directed architecture. An FHP computer appears as a FORTRAN machine to a program written in FORTRAN, or as a COBOL machine to a program written in COBOL. High-level language statements are compiled into S-language statements, which an FHP processor directly interprets using dynamically switched, S-language interpreters. FHP systems use these S-language interpreters to change a processor's interface depending on which S-language is running. Most FHP systems' S-interpreters will reside in fast semiconductor control store to minimize the time needed to switch interpreters.

## Addressing Flexibility

A high-level architecture should provide a large virtual memory. The architecture's addressing mechanisms should place no constraints on the size of user programs. Addressing mechanisms should also allow controlled sharing of stored data. The high-level language programmer should be isolated from these addressing mechanisms. This means that architecturally-supplied system functions must manage address translation and memory allocation.

All FHP systems share a two-dimensional, 2**112-bit (2**109 8-bit byte) address space. This is the largest address space currently available to data processing users. FHP systems manage this memory without programmer intervention. You have controlled access to data stored anywhere in an FHP system's memory hierarchy (such as main memory or disk), without knowing the data's location.

The FHP architecture manages this enormous address space by dividing it into uniquely named and protected information containers called objects. Objects are the FHP architecture's basic unit of storage. The two-dimensional, address space contains 2**80 objects as one dimension. Each object can contain up to (2**32)-1 bits (512 megabytes) of information (the address space's second dimension). Each object is uniquely named by an 80-bit Unique Identifier (UID). Because objects are uniquely identified, an FHP program can address any bit in any object on any FHP system. Each object has a set of attributes that describes the object's length, type, and access control information.

There are two kinds of object: primitive and extended-type. Primitive objects include data and procedure objects. Programmers can create Extended-type Objects, and define what operations can be performed on them.

FHP systems use Namespace Addressing to automatically reference S-language instruction operands. FHP's Namespace Addressing eliminates the need for programmers to explicitly access a processor's hardware registers. FHP S-language instructions refer to operands only by name. Namespace Addressing automatically translates operand names into operand descriptors. The system uses these descriptors to automatically calculate an operand's location, length, and type.

Namespace Addressing supports consistent and efficient high-level language processing, with guaranteed program and data protection. Namespace Addressing also allows hardware or firmware enhancements that do not affect a high-level programmer's view of an FHP system. FHP programmers need not be concerned with register management or optimization.

## Security

A secure architecture should supply efficient protection mechanisms that check every memory reference against a user's right to make the reference. The FHP architecture provides flexible access control and containment mechanisms to protect programs and data. These powerful security mechanisms let programmers create data and program protection systems to fit specific applications.

Each FHP object has an Access Control List (ACL) that contains the information needed to validate each reference to the object. An object's ACL has an entry for every user who has access to that object. An ACL entry specifies which operations the user can perform on the object. A user cannot reference an object if his name is not in the object's ACL. Primitive objects support read, write, and execute operations, as well as non-data mode operations. Non-data mode operations let authorized users read or modify an object's attributes. Extended-type Objects have Extended Access Control Lists (EACLs) that let programmers define more complex access privileges. Extended-type Objects also support non-data mode operations.

FHP's security system also implements domains, which let programs specify a process' authority. Domains provide controlled access to groups of programs. A user working in one domain can be assigned different access privileges when his process executes procedures in other domains. A user's original access privileges are restored when he returns to the first domain. FHP's security system uses an Architectural_Call function that checks the validity of any program-to-program or program-to-system call. The system guarantees secure returns from calls by protecting the caller's state information.

## Technological Growth Potential

Current trends show that computer hardware costs are declining, while the cost of producing software is increasing. Many kinds of extremely fast but inexpensive Random Access Memories (RAMs) are now available. A well-designed architecture should use these state-of-the-art components effectively, and adapt to future developments in hardware and software.

Many conventional systems take limited advantage of technological advances in semiconductor memories; their architectural limitations were established before fast and inexpensive logic components were available. Taking numbers from two registers, adding them together, and storing them are easy tasks for today's components. However, these components can perform much more powerful operations in the same amount of time, at essentially the same cost. Conventional architectures can take advantage of lower

memory costs by adding more semiconductor main memory. However, because of limited logical address space, main memory expansion often does little to increase system performance.

Data General's architectural developers took full advantage of advances in semiconductor technology to make FHP a memory-intensive architecture. FHP uses system tables and data structures that can be easily accelerated by hardware or firmware. Acceleration means that you move frequently used data into more accessible locations. Acceleration techniques include overlapping data and instruction fetches (pipelining), hash-coded table searches, and cache storage. A cache stores data that a processing function is likely to use or re-use. Conventional architectures can use programmer-managed registers as accelerators, but caches perform the same acceleration with greater flexibility, and without explicit programmer intervention.

The FHP architecture will also keep pace with advances in high-level programming languages. Most conventional systems are language-dependent because they support some high-level languages better than others. This is why some computer systems are considered "FORTRAN" or "COBOL" machines. FHP's language-directed architecture will serve future high-level languages as efficiently as it serves present languages.

This completes our introductory look at the factors that influenced the design of Data General's new FHP Architecture. We also introduced some of the architecture's features. In the next two chapters, we describe these features in some detail, and show how they are integrated into the architecture. We define new terms as they are introduced. Please consult the Glossary (Appendix A) if you are unsure about a term's meaning.

--End of Chapter--

# Chapter 2
# Features of the FHP Architecture

In Chapter 1, we introduced the FHP architecture's design goals and five major FHP features:

1) Sets of machine-language substitutes, called S-languages, that support FHP's language-directed architecture.

2) A 2**112-bit, universally addressable memory, divided into uniquely identified information containers called objects.

3) Flexible and efficient instruction-stream operand references using Namespace Addressing.

4) Architecturally supplied protection mechanisms that use Access Control Lists to control user's rights to objects.

5) A memory-intensive design that allows easy acceleration of common system functions.

This chapter outlines the first four FHP features, and shows how FHP's memory-intensive design is used to increase the speed of basic architectural functions. To demonstrate FHP's power and flexibility, we compare its features to those of conventional systems.

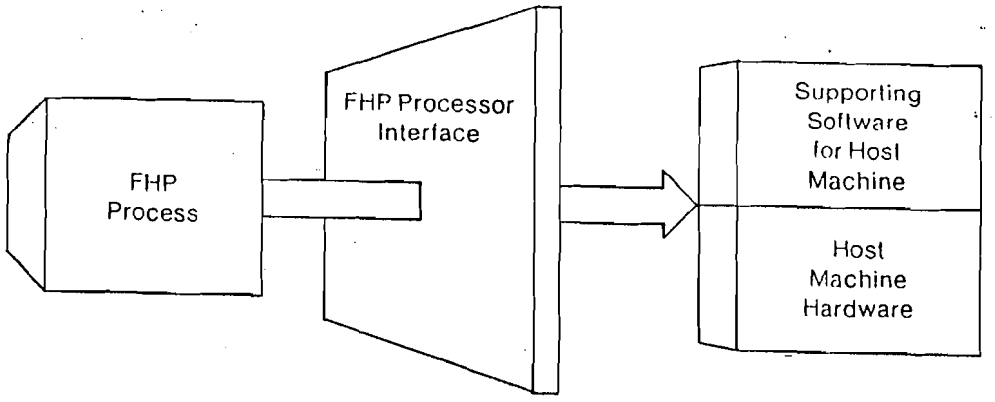
Before beginning our discussion, we review some terms used in the rest of the manual:

* Architecture
* Program
* Process
* Processor
* Host machine

An architecture is a guaranteed user interface to a system. FHP consists of a collection of architectural features that contribute to FHP's overall structure. Because of FHP's modularity, we can isolate features to show how they help execute high-level language programs.

A program is a series of instructions that controls a computer's operation. To ensure that system resources are used

OK




TC-00024-00

2-1

CAPTION: FHP processes see the same processor interface, regardless of a system's hardware or software implementation.

efficiently, programs are executed by operating system data structures called processes. Programs are static; they do no work until they are executed by a process.

A process represents a computer system's state as the system executes a program (or group of programs). A process performs work for a specific user. A computer system's operating system determines what resources an executing program requires and makes those resources available on a shared basis. These resources include actual Central Processing Unit (CPU) time and address space allocation.

A process maintains the state of the system while a program is running. By saving this state information, the operating system can stop a process, then restart it later by restoring its state. If a program makes an I/O request, its controlling process can be suspended to let another process run.

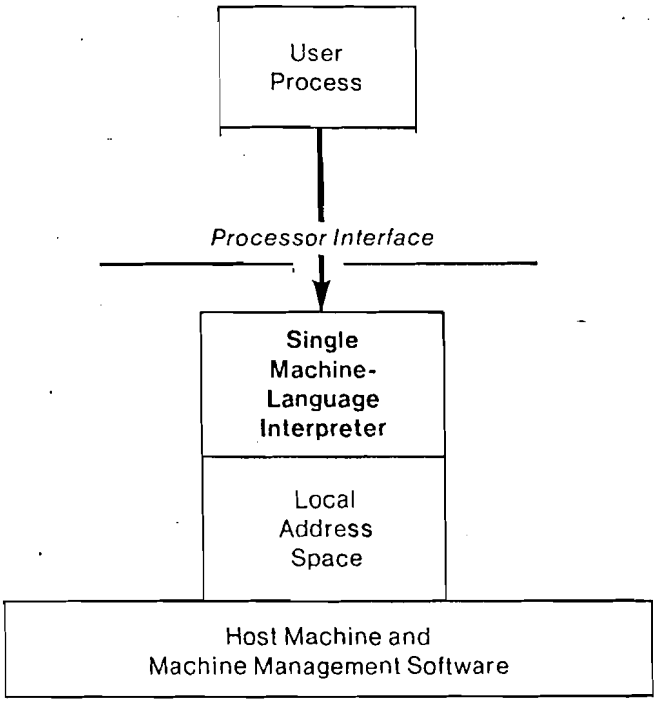
A process runs on a processor. FHP does not limit the number of processors a model can have, nor restrict the way the processors are implemented. The architecture does require that all processors present the same appearance to a high-level language programmer, so that he appears to have his own, independent FHP processor. Because software is inherently more flexible than hardware, FHP processors will usually be implemented using a combination of hardware and software.

Each user actually shares a system's host machine with all other system users. An FHP system's host machine is a collection of hardware that is enhanced with software and firmware to create a set of FHP processors. A processor that is implemented with a combination of hardware and software is called a Virtual Processor (VP). We discuss process-to-Virtual Processor scheduling in this chapter's last section.

tc24

Figure 2-1. Host Machine and Supporting Software

The architecture permits a wide range of FHP models, using different combinations of host machine hardware, firmware, and software. Selection of an FHP model's host machine configuration is based on cost/performance considerations. High-performance FHP systems can move more processor functions into host machine hardware, and can even use multiple host machines. Smaller FHP systems can move the same functions into software, to reduce hardware costs.




TC-00019-00

2-2

CAPTION: Conventional systems generally use a single, general-purpose machine language. They also limit processes to small, localized address spaces.

TC19

## The FHP Processor

The FHP architecture is best understood by looking at the interface an FHP processor establishes for a process. Multiprocessing FHP systems provide one or more sets of identical FHP processors, and any process can run on any processor. To keep our presentation clear, we discuss how a single FHP process views each architectural feature. We also introduce some non-architectural techniques that models of the architecture can use to accelerate selected functions. Simply stated, an FHP processor:

* interprets and executes S-language programs; and
* provides controlled access to objects in FHP's universal address space.

The following sections describe how an FHP processor performs these functions, and show the difference between conventional and FHP systems.

## 2.1 S-languages and S-language Interpreters

The FHP architecture replaces a conventional system's single machine language with a set of special-purpose languages called S-languages. Each S-language is tailored to the unique requirements of a different high-level language. S-languages are much more efficient than conventional machine-language instructions. Before describing the power of FHP's S-languages, we review the way conventional machines perform operations and reference operands.

### Conventional Machine Languages

Conventional systems usually have one, general-purpose, machine-level language, containing a set of about 200 instructions. This single instruction set must support the requirements of all high-level languages that will ever run on that system. Figure 2-2 illustrates a conventional system's processor and its single general-purpose, machine-language interpreter.

tc19

Figure 2-2. A Conventional Processor

Systems with general-purpose instruction sets often force programmers to develop programs in a machine-efficient language, rather than an application-efficient language. This increases the

cost of preparing and maintaining application software. In many situations, your application might best be served by two high-level languages. You might use a commercial language for generating reports and a scientific language for data calculations. If you ran this application on a conventional processor, programs written in a language like FORTRAN might use your system's resources efficiently. However, programs written in COBOL might use the system resources inefficiently. Because of this high-level language dependency, many computers are considered to be FORTRAN or COBOL machines.

A conventional machine-language instruction conveys the following information to the processor:

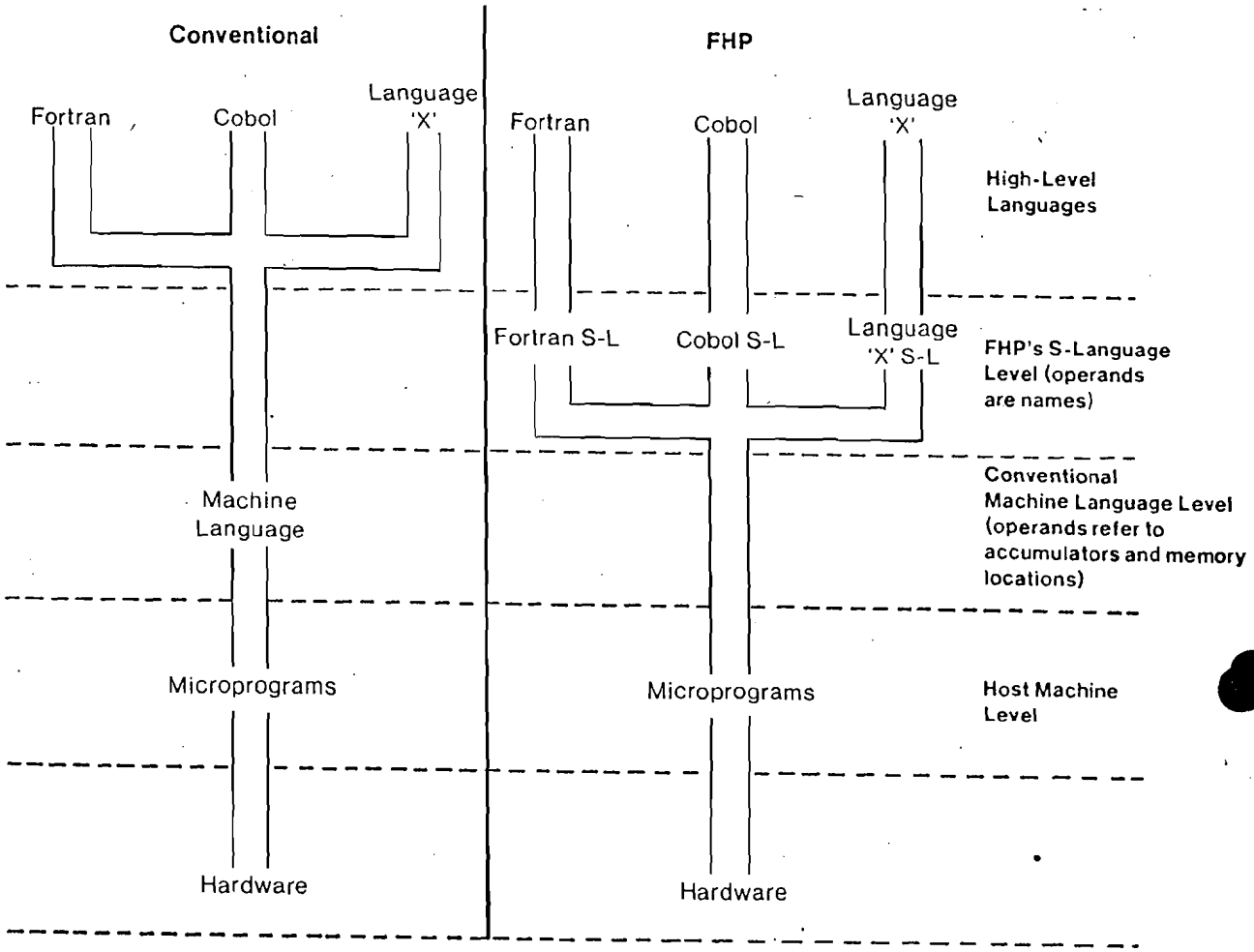
* The operation to be performed, such as add or subtract
* The number of operands in the instruction
* A description of each operand: its location, length, type, and direction of data flow (read or write)

A conventional architecture's operation code (opcode) usually provides all of the above information except an operand's location. Conventional architectures generally limit the number of operand types (e.g., real, floating) and the number of operands in an instruction (typically one, two, or three). An operand's length is usually fixed at 8, 16, or 32 bits.

The machine-language interface to a conventional microprogrammed processor can be changed by modifying the host-machine's microcode to support new instruction sets. These "soft" machines are more flexible than "hardwired" instruction set machines. However, you must usually remove the system from service to load a new machine-language interpreter into control store from a secondary storage device. FHP removes these limitations by replacing the single general-purpose machine language with a set of machine-language substitutes, called S-languages.

## FHP S-languages

S-languages are intermediate languages that have semantic content between that of conventional assembly languages and high-level languages. Semantic content is a measure of how much processing work results from a single instruction. S-languages refer to operands by name only and do not reference any host machine registers or accumulators.




TC-00025-00

2-3

CAPTION: Conventional systems usually have a single, general-purpose machine language. Compiler writers must develop compilers that force-fit this machine language to the requirements of different high-level languages.

The FHP architecture supports multiple machine-language substitutes, called S-languages. Each S-language is tailored to the requirements of one or more high-level languages.

FHP systems deal with an S-language instruction's opcode separately from the instruction's operands. S-language interpreters examine S-language opcodes and tell the host machine to perform the specified operations. FHP's Namespace Addressing mechanism is responsible for referencing S-language instruction operands, regardless of which S-language is executing. We describe FHP's dynamically switched S-language interpreters and Namespace Addressing later in this chapter.

S-languages let Data General system programmers define a small set of tailored instructions (S-instructions) for each high-level language that runs on an FHP system. Essentially, FHP places no architectural limit on the number of S-languages a system can support, and each S-language can have as many as 256 different operation codes. FHP systems need never "force" machine instructions to fit a number of different languages; an S-language always closely matches the semantics of the high-level language it supports. Of course, an S-language instruction can be used by a group of high-level languages that have similar processing requirements.

S-languages accomplish more processing work for each instruction than do conventional machine-language instructions. Figure 2-3 compares the semantic content of FHP's S-languages to that of high-level and conventional assembly languages.

tc25

Figure 2-3. Comparison of Computer Language's Semantic Intensity

An FHP S-language's high semantic content provides distinct performance advantages over conventional machine languages. A high-level language statement which compiles into ten machine language statements might require only two S-language statements. Therefore, for most high-level language programs, FHP systems have to fetch, examine, and execute fewer instructions than conventional systems do.

The following simple examples show the close match between typical FHP S-languages and high-level languages such as FORTRAN and COBOL.

| FORTRAN | FHP FORTRAN S-language |
|---|---|
| I = J + K + L | IADD J, K, I<br>IADD2 L, I |
| A = B(J,K) +C(L) | FADD B (base J,K), C (base L), A |

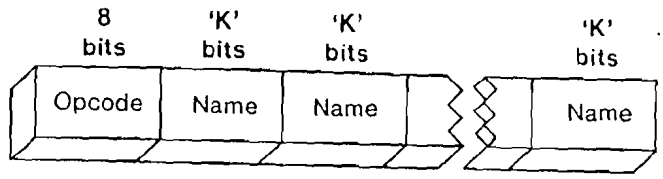
| COBOL | FHP COBOL S-language |
|---|---|
| IF A = B THEN<br>ADD A TO C<br>GIVING D | DBE A, B, BR<br>DADD2 A, C, D |

The typical S-language mnemonics shown above are almost self-explanatory. The FORTRAN S-language example illustrates two integer add instructions (IADD and IADD2) and one floating point add instruction (FADD). IADD adds the first variable to the second and places the result into the third variable. IADD2 adds the first variable to the second and places the result into the second variable. The FADD instruction is the same as the IADD instruction, except FADD operates on floating point values.

The COBOL S-language example illustrates a branch instruction (DBE), and a decimal add instruction (DADD). DBE compares the values of the first and second variables. The value in the third operand is added to the program counter's value if the first two variables are equal. If the first two variables are not equal, execution continues in sequence. The DADD instruction adds the value of the first variable to the second, and places the result into the third variable.

A typical S-language instruction set is very compact and efficient because an instruction's addressing mode information is not carried with the instruction. An FHP system that supports FORTRAN and COBOL might have approximately 70 S-instructions for FORTRAN, 100 for COBOL, and another 70 for a high-level system programming language. Together, these three languages have about 240 instructions--more than in the instruction set of many conventional machines.

FHP S-languages provide an excellent tool for compiler writers and system programmers because of the comparatively small semantic span between a high-level language and its associated S-language.




TC-00011-00

2-4

CAPTION: All S-language instructions use an 8-bit opcode. The opcode is followed optionally by any number of k-bit operand syllables. 'k' can be either 8 or 16 bits.

TC11

Conventional assembly language instructions refer to both addresses and registers. The compiler writer must manage this information manually, which can introduce errors into compiler code. S-languages reference only operand names because every S-language uses Namespace Addressing to reference operands. Therefore, compilers written in FHP S-languages can be more reliable than those written in conventional assembly languages because S-languages are easier for compiler writers to use.

To add a new high-level language to a conventional system, you must compile to that system's machine-language instruction set. This fixed instruction set may not provide efficient interfaces to your new language. When we add another language to an FHP system, we can create a new S-language tailored to the high-level language. The new S-language does not affect existing ones. FHP's S-languages are independent of one another; interpreters for each S-language are dynamically switched as a process executes different source languages.

Some models of the FHP architecture will allow User Accessible Microprogramming (UAM), so that you can prepare your own S-language interpreters in host machine microcode.

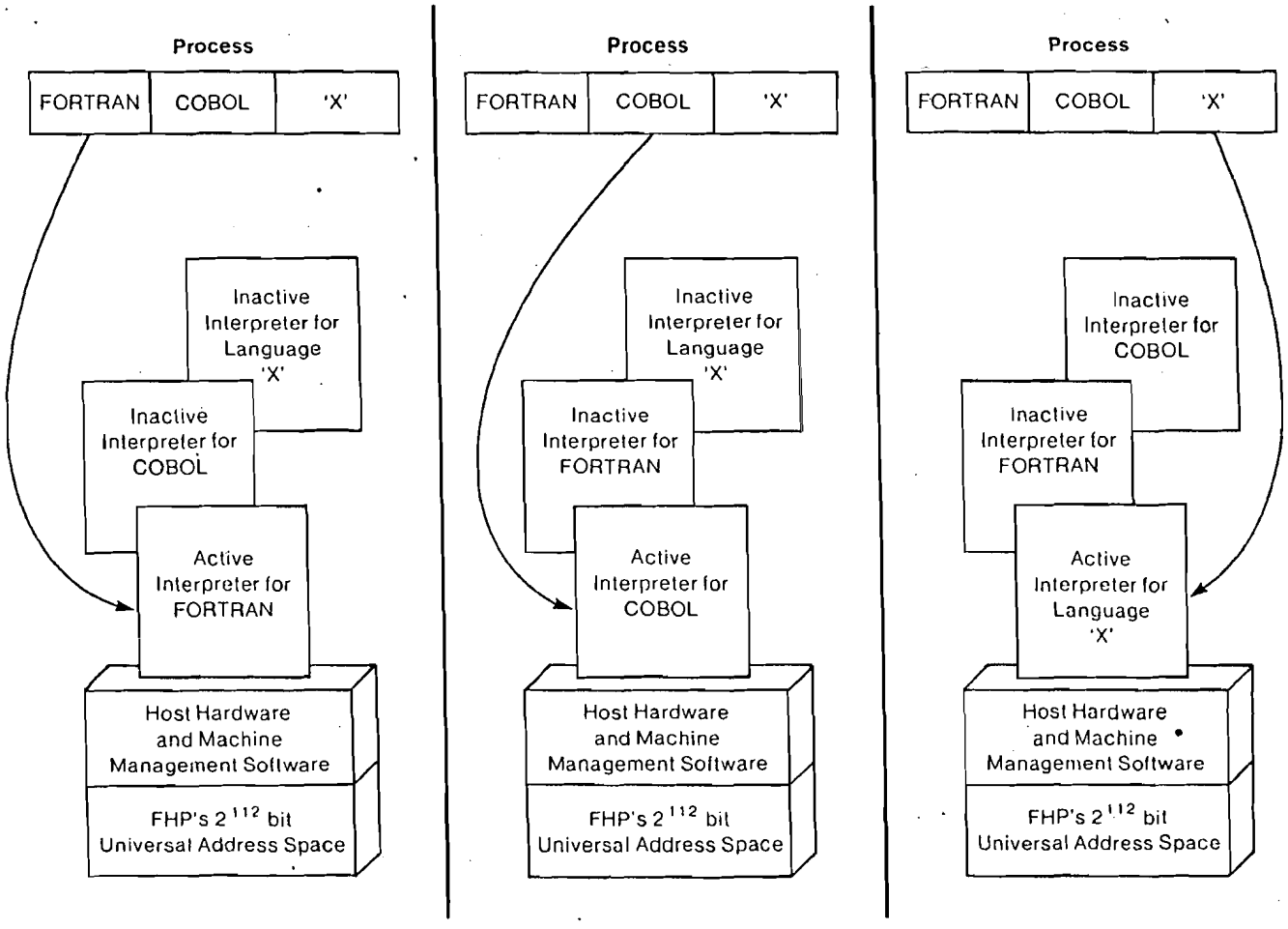
### 2.1.1 S-language Instruction Syntax

All S-langugages use the same instruction stream syntax. An S-language instruction is made up of an 8-bit operation code (opcode) syllable, followed optionally by some number of operand syllables. See Figure 2-4 below.

tc11

Figure 2-4. FHP S-language Instruction Syntax

An FHP operand syllable can directly represent a literal value, or indirectly represent an operand. Literal values can be used as instruction stream branch offsets. A syllable that indirectly represents an operand is called an operand name. Operand names are numbers of size 'k', where 'k' is either 8 or 16 bits. Operand names can indirectly describe many kinds of data, such as scalars, vectors, arrays, bit strings, and pointers to other locations.




TC-00014-00

2-5

CAPTION: FHP systems automatically switch S-interpreters when a process runs different S-language programs. Each S-language program runs on a machine that is ideally suited to the program's processing requirements.

## 2.1.2 Dynamically Switched S-language Interpreters

High-level language programs on FHP systems are compiled (translated) into their corresponding S-language programs. An FHP system automatically activates the appropriate S-language interpreter (S-interpreter) when the system calls a program using a different S-language.

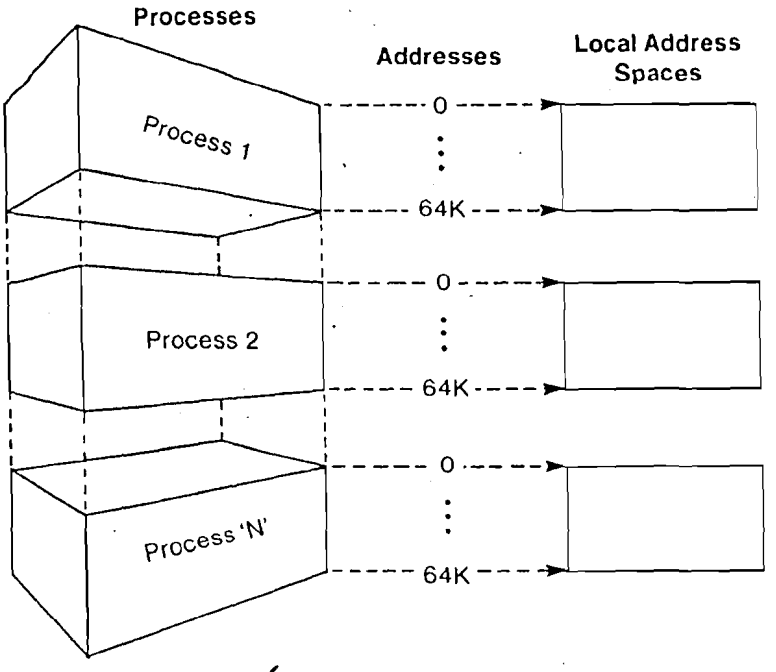
An FHP S-interpreter receives and examines an S-language instruction's operation code. The S-interpreter then tells the host machine to perform the operation specified in the instruction's operation code. S-interpreters provide an interface between a system's S-languages and the host machine. High-level language programs that are compiled into Data General S-languages will run on different FHP systems, even if the systems have different host machine hardware. FHP S-interpreters will usually be written in host machine microcode and will reside in fast control store when activated by a processor. Inactive S-interpreters can be stored in a system's main or secondary memory and moved into control store on demand.

An FHP model's operating system dynamically switches S-interpreters as a process executes programs written in different high-level languages. Figure 2-5 illustrates how the interface to an FHP processor is switched as a process runs programs written in three different source languages. Compare this figure to the conventional system shown in Figure 2-2 above. The FHP system's FORTRAN S-interpreter is active while the FORTRAN S-language program is executing. The system automatically activates a COBOL S-interpreter when a COBOL program is called, and activates the third language's S-interpreter when that source language is called. Therefore, the FORTRAN program "sees" only a FORTRAN processor and the COBOL program sees a COBOL processor.

tc14

Figure 2-5. Dynamically Switched S-interpreters

TC-00026-00




TC-00026-00 2-6

CAPTION: Processes on conventional systems usually have small, localized address spaces. Address information cannot be passed from one process to another, because all processes issue the same set of logical addresses.

## 2.2 Uniquely Named Object Addressing

In the last section, we told you how FHP uses dynamically switched S-language interpreters to eliminate many language dependencies of conventional systems. However, conventional addressing mechanisms often do not match a high-level language's addressing requirements because they have one or more of the following restrictions:

* a limited address space for each user's process.
* an address space that is local to one process.
* the inability to efficiently support the complex addressing needs of high-level languages.

The FHP architecture provides solutions to each of these problems. This section explains how FHP solves the first two problems with its universal, object-oriented address space. The next section explains how Namespace Addressing supports complex addressing requirements.

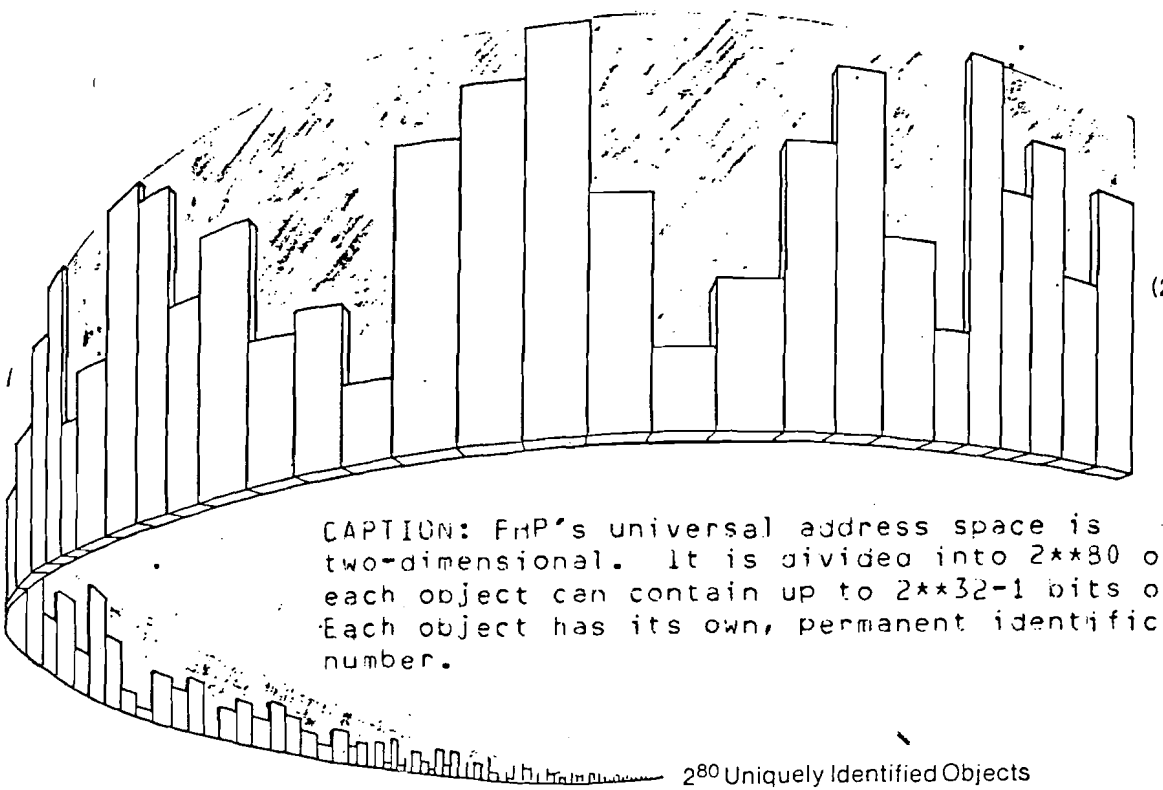
### Addressing in Conventional Systems

Many conventional architectures limit the logical address space available to system and application programmers. This small address space, 64 to 128K bytes on many computers, is adequate for many applications and can be circumvented in others. However, transferring large application programs from mainframe computers to smaller machines is often a problem. Programmers must divide large programs into smaller pieces, called "overlays." This technique is often time-consuming and unreliable. Further, the operating system must use processor resources to swap these overlays to and from main memory. An efficient architecture must have an address space large enough so that programmers need not artificially restrict program or data structure sizes.

Conventional systems usually have multiple local address spaces. Each process has one, contiguous address space, independent of other processes' address spaces. Each process issues the same addresses as all other processes. These addresses are not unique. This constraint makes it difficult to share information between processes. See Figure 2-6 below.

tc26

Figure 2-6. Address Spaces in a Conventional System




CAPTION: FHP's universal address space is two-dimensional. It is divided into 2**80 objects, and each object can contain up to 2**32-1 bits of data. Each object has its own, permanent identification number.

Lastly, high-level programming languages have complex storage requirements that differ from the rigid storage methods provided by conventional machines. High-level languages often deal with complex structures, such as pointers and arrays of records. Conventional architectures, however, place limitations on an operand's size and type, and the way operands can be addressed.

High-level languages also reference variables and data structures by name. Named variables and structures differ in size and type from language to language. Conventional machine languages have no notion of objects, but do have distinct instructions for different variable's types (integer, floating point, etc.). The gap between the addressing requirements of high-level languages and conventional addressing mechanisms make it difficult for these systems to process high-level languages efficiently. FHP's extremely flexible addressing mechanisms offer significant advantages over conventional addressing.

## FHP's High-Level Addressing

FHP's architecture eliminates conventional address space restrictions by providing a 2**112-bit (2**109 byte), universal address space. Universal means that FHP's address space is available to any process on any FHP system. FHP systems reference data in this universal address space with two addressing steps. The basic addressing mechanism references uniquely-named objects, and is called UID Addressing. A higher-level addressing mechanism, called Namespace Addressing, references instruction stream operands within individual objects. We describe these two FHP addressing mechanisms in the following two sections.

## Uniquely Named Objects

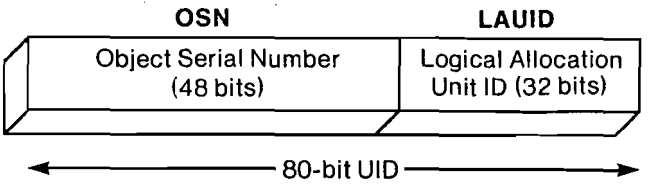
FHP's logical address space is divided into 2**80 uniquely named, protected information containers called objects. Each of the 2**80 FHP objects can contain from zero to (2**32)-1 bits (512M bytes) of data. FHP's universal address space is two-dimensional and contains 2**80 times 2**32-1, or 5.19x10**33 bits. A representation of this address space is shown in Figure 2-7 below.

tc3

Figure 2-7. FHP's Universal Logical Address Space

Each FHP object has a Unique Identifier (UID) that is used to locate an object in an FHP system's memory hierarchy: in main

OK




TC-00007-00

2-8

memory, on disk, or even in another system. UIDs are unique across all FHP systems; when an object is deleted from a system, its UID is never used again.

## Unique Identifiers

FHP's UIDs are designed for use in distributed processing environments. Each object's 80-bit UID has two components:

* a 48-bit Object Serial Number (OSN)
* a 32-bit Logical Allocation Unit Identifier (LAUID)

tc7

Figure 2-8. Basic UID Structure

Data General assigns part of the 32-bit Logical Allocation Unit Identifier (LAUID) to a system, to guarantee that UIDs remain unique from system to system. Another part of the LAUID identifies logical units within a system. Logical units contain maps which the system uses to locate objects on physical devices, like disk drives. Objects residing on a logical unit have the same LAUID component, but will have unique Object Serial Numbers.

Each Logical Allocation Unit has a directory that contains the attributes and logical location of each object in the unit. An object's attributes include information about the object's type (whether it contains procedures or data), its size, and information used to protect the object (its Access Control List). Access to information in a Logical Allocation Unit Directory (LAUD) can be accelerated with techniques outlined in Chapter 3.

## How UIDs Are Generated

The FHP architecture does not specify how UIDs are generated. The architecture only requires that UIDs be unique. However, most FHP systems use the FHP Architectural Clock to generate the 48-bit OSN component of UIDs. This clock "ticks" about every 0.23 nanoseconds, and is considered to have started on April 15, 1968. Different models of the FHP architecture will generate UIDs at different rates, depending on host machine hardware cycle times. No FHP system can generate UIDs faster than the base rate of the Architectural Clock, however.

The FHP SPRINT (the first model of the FHP architecture) can generate a new UID approximately every 10 microseconds by incrementing the 48-bit OSN. An operating system request for a new UID reads the system clock and uses this reading to produce the OSN component of the UID. UIDs generated at this rate will remain unique until at least the year 2080.

### 2.2.1 FHP Objects

The FHP architecture has two basic object classifications: primitive and extended-type. Primitive objects allow a small set of architecturally defined and controlled operations. Extended-type Objects, however, let the programmer define and control the operations performed on them. Primitive and Extended-type Objects are described in the next two sections.

#### 2.2.1.1 Primitive Objects

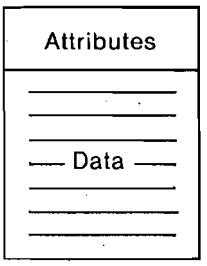
You can perform both data-mode and non-data-mode operations on primitive objects. Data-mode operations are limited to read, write, and execute. Non-data-mode operations let you manipulate the primitive object's attributes and Access Control List.

FHP defines four kinds of primitive object: procedure, data, S-interpreter, and Extended-type Manager.

Procedure objects

Your FHP system's high-level language compilers or binder creates procedure objects for you. A procedure object contains executable code in the form of S-language instructions. Procedure objects also contain literal data, and information indicating which S-interpreter will process the procedure's instructions. Each procedure object contains a table of operand descriptors which Namespace Addressing uses to reference an S-language procedure's operands. We describe this Name Table in "Namespace Addressing," later in this chapter.

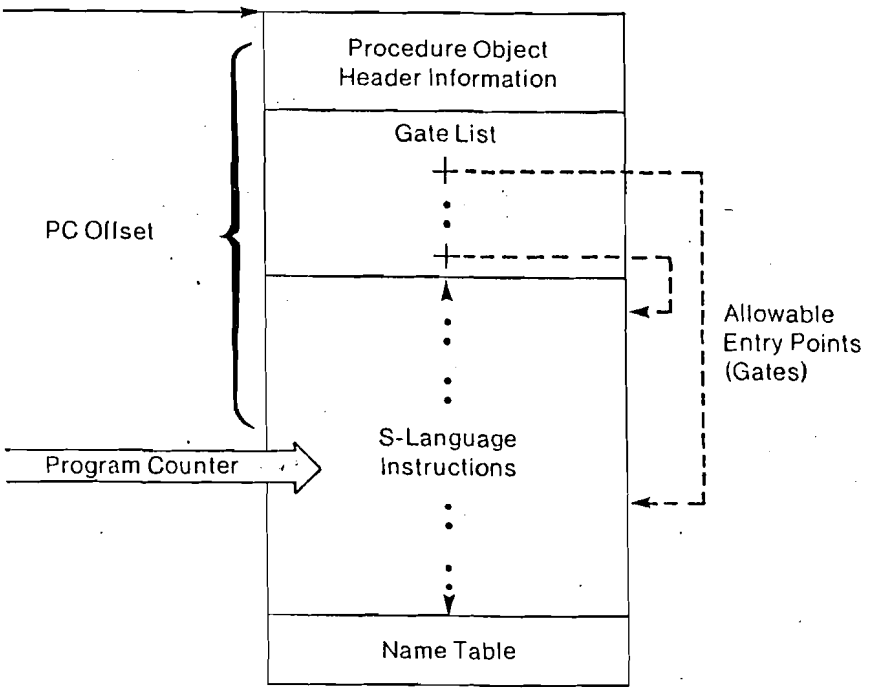
Procedure objects can contain more than one S-language procedure. Each procedure object has an information header, followed by a list of entry points, or "gates," into the object (See Figure 2-9 below). Gates help control a caller's access to a procedure by preventing calls that try to start in the middle of a procedure. The procedure object's gate list contains logical addresses of legal entry points into each procedure in the object. The system rejects calls that do not reference a legal gate list entry. The logical addresses in the gate list are in special data structures called pointers. We describe FHP pointers later in this




TC-00027-00

2-10

OK




TC-00016-00

2-9

CAPTION: A typical FHP procedure object's header contains information about the rest of the object. The gate list contains the locations of legal entry points into the S-language procedure. The Name Table contains information that Namespace Addressing uses to calculate the logical addresses of the procedure's variables. A process uses the 32-bit Program Counter to locate the currently executing S-language instruction.

TC11

chapter.

Each procedure in a procedure object is identified and located by the procedure's Entry Descriptor (ED). A procedure can be called from outside the object if its ED is in the object's gate list. Otherwise, the procedure can only be called from within the object. Each procedure is associated with a Procedure Environment Descriptor (PED). The system uses information in the PED to prepare the procedure's environment. Procedures with the same environmental requirements can share a PED. A PED includes pointers that locate the procedure's S-interpreter, Name Table, and data storage areas.

tc16

Figure 2-9. A Simple Procedure Object

Each FHP process uses the S-language Program Counter (PC) to locate the currently executing S-language instruction. The PC contains a 32-bit offset, which points to the currently executing S-language instruction's opcode. The Program Counter's offset value is saved as part of a process' state information. The PC's contents can be modified with S-language branch instructions. We explain branching in "Namespace Addressing," later in this chapter. The value of the S-language Program Counter is saved as part of a process' state.

## Data Objects

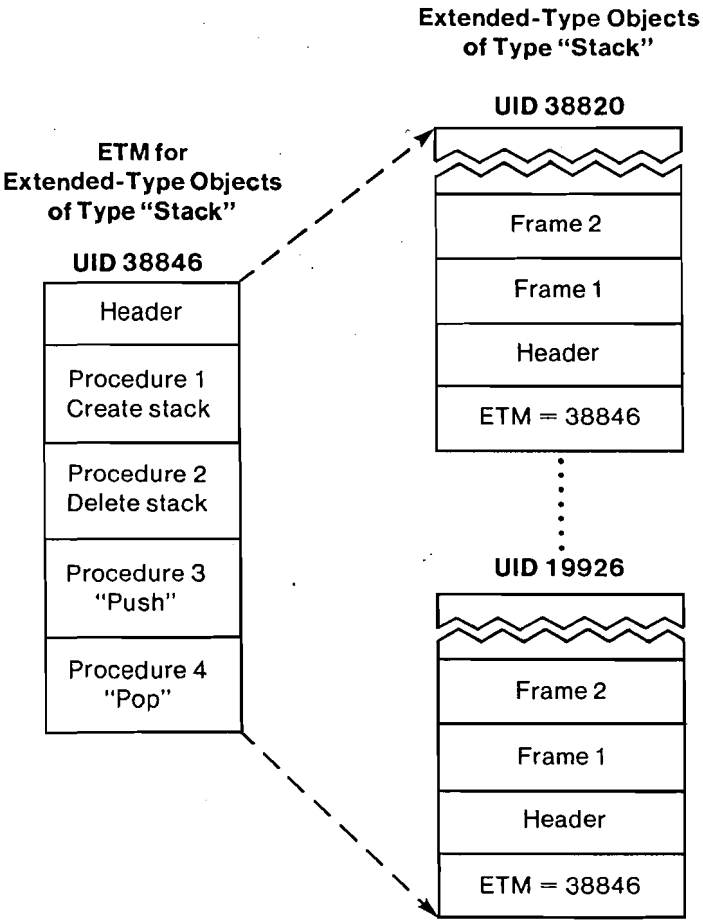
Data objects simply contain data; they have no information header or gate list. Data objects are protected with an Access Control List (ACL), like all other FHP objects. Programmers can set a data object's ACL to prevent attempts to execute the object, while controlling read and write access.

tc27

Figure 2-10. A Typical Data Object

## S-interpreter Objects

S-interpreter objects are similar to procedure objects, but they contain microcode that examines an S-language instruction's




TC-00052-00

2-11

opcode and directs the host machine's operation. An FHP system will have an S-interpreter object for each S-language that the system can process. One S-interpreter may serve many high-level languages with similar requirements, or one high-level language program can be executed using different S-interpreters.

Extended-type Managers

Extended-type Managers (ETMs) are also similar to procedure objects, but they establish a user-defined interface to one or more Extended-type Objects. ETMs contain procedures which perform operations on Extended-type Objects. We describe the relationship between ETMs Extended-type Objects in the following section.

### 2.2.1.2 Extended-type Objects

One of FHP's most powerful features is its ability to create user-defined Extended-type Objects. Extended-type Objects let the user define operations beyond primitive read, write, and execute operations. Programs that execute extended operations are contained in an Extended-type Object's ETM. An Extended-type Object may also have an Extended Access Control List (EACL) to control access to its extended operations. We describe Extended-type Object protection in "FHP's Security System," later in this chapter.

An example of an Extended-type Object is a stack. A stack is a contiguous section of storage that lets you store data by "pushing" it onto the stack. You retrieve this data by "popping" it off the stack. FHP programmers can create protected stack objects by defining a stack_manager ETM. This stack manager might contain procedures that let you create and delete extended-type stack objects. The stack manager would also contain procedures that implement the push and pop operations.

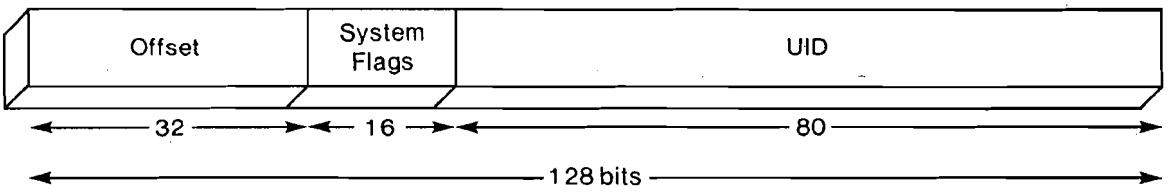
Each Extended-type Object names its ETM by referring to its ETM's UID. However, a reference to an Extended-type Object is made using the Extended-type Object's UID. Figure 2-11 below shows an Extended-type Manager that has procedures to create and delete Extended-type Objects of type "stack." This stack manager also contains procedures which support the push and pop operations. Notice that one ETM can support more than one Extended-type Object.

tc52

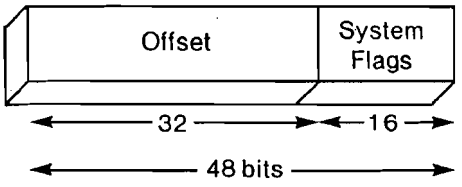
Figure 2-11. Extended-type Manager and Extended-type Objects

OK

**Inter-Object Pointer**



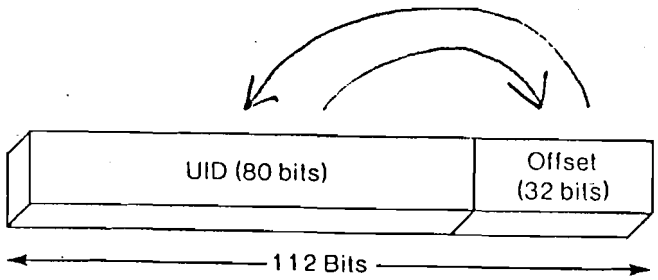

**Intra-Object Pointer**




TC-00004-00

2-141

OK

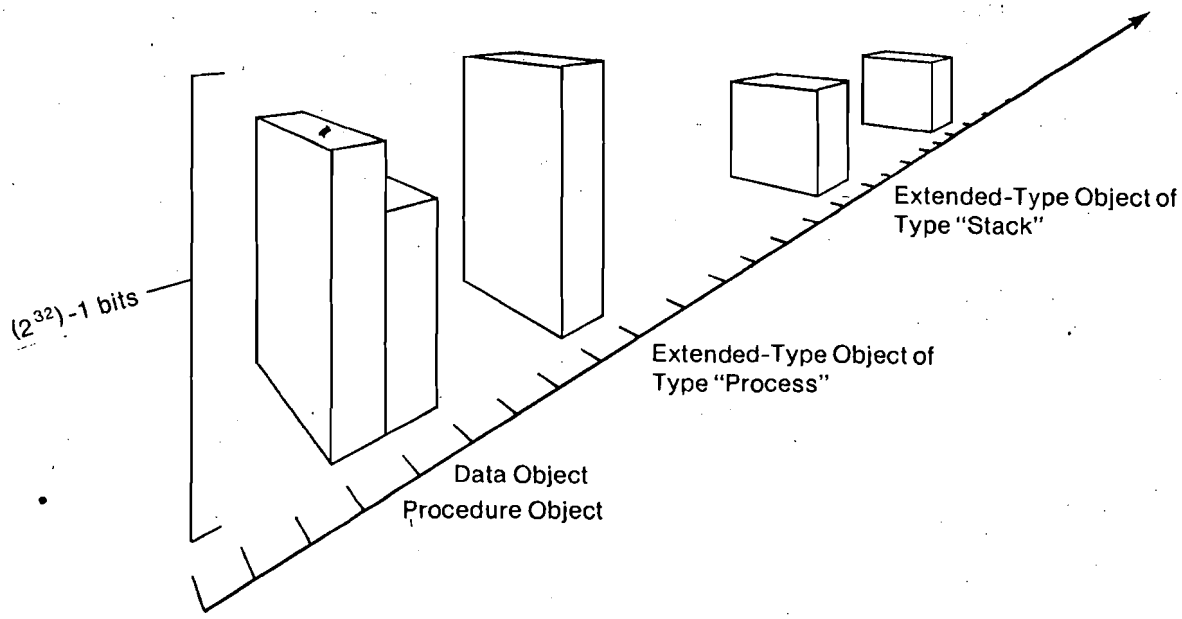



TC-00012-00

2-13

CAPTION: All FHP programs use 112-bit logical addresses. The 80-bit UID field locates an FHP object; the 32-bit offset field locates a bit within the object.

TC12




TC-00002-00

2-12

An FHP operating system can define and use several kinds of stack objects to support functions such as process management. FHP systems also use the Extended-type Object and Extended-type Manager facilities to create and manage Extended-type Objects of type "process." An FHP operating system can manipulate processes like any other object. Figure 2-12 below shows these different kinds of objects in the FHP address space.

tc2

Figure 2-12. Types of FHP Objects in the Address Space

## 2.2.2 FHP's Logical Address and Pointers

All FHP programs use 112-bit logical addresses, which span FHP's entire, 2**112 bit address space. FHP's logical addresses have two parts: one that specifies an object's UID, and another that locates a bit in the object. The UID part of a logical address is 80 bits; a bit in the object is located with a 32-bit offset. For example, an FHP logical address of [3625,17072] specifies bit number 17072 in object number 3625.

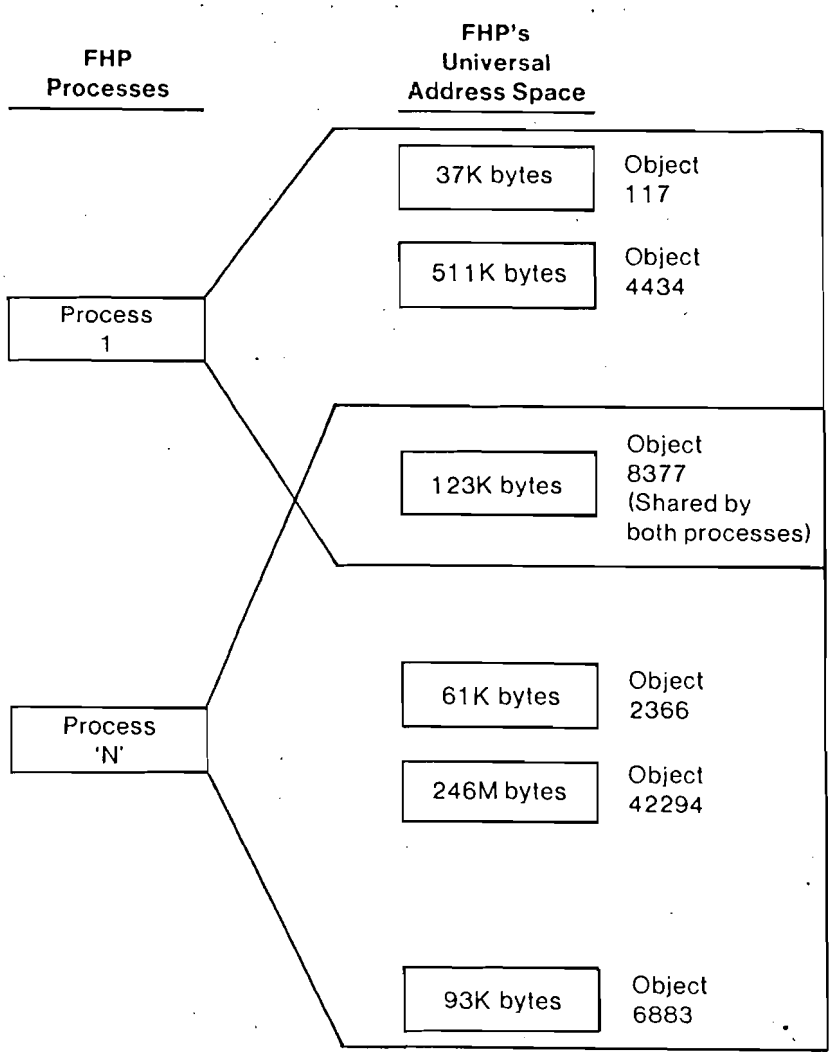
tc12

Figure 2-13. FHP's Logical Address

An FHP system's data structures can be located using two kinds of architecturally defined pointers: inter-object and intra-object (see Figure 2-14 below). A pointer's flags and format field specify the pointer's type. Pointers also provide non-architectural fields which may be used by S-language interpreters. The combination of UID and offset (logical address), plus 16 bits of system-defined information (flags and format) is called an inter-object pointer. FHP systems use inter-object pointers to name any bit in FHP's universal address space. The 48-bit intra-object pointer references data and procedures within the object containing the pointer.

tc4

Figure 2-14. FHP Pointers




TC-00096-00

2-15

CAPTION: All FHP processes share FHP's universal address space. A process' address space is determined by the process' requirements, not by artificial adures-sing boundaries. Processes can easily share the data in one or more objects.

FHP pointers are either resolvable or non-resolvable. Resolvable pointers contain an FHP logical address in the form of UID and offset. An FHP system's Namespace Addressing mechanism can operate on resolvable pointers directly.

One kind of non-resolvable pointer can be converted to a resolvable pointer by an operating or language system at runtime. This feature lets FHP programmers delay the linking of a group of objects until a program referencing the objects is run. This is called dynamic linking. Dynamic linking lets programmers make references to an object by using a non-resolvable pointer, which refers to the object's symbolic name. The relationship between the object's symbolic name and the object's logical address (UID and offset) is established by the system's language or operating system. The system recognizes a non-resolvable pointer by examining the pointer's resolvable/non-resolvable flag.

## FHP's Address Space

Inter-object pointers are unique even when passed from process to process or system to system. This is in contrast to conventional systems, where logical addresses are meaningless when passed from process to process. Conventional processes issue the same set of logical addresses, which are translated to physical locations when the process runs. FHP processes issue unique logical addresses that other processes can use to share data. (Compare Figure 2-6 with Figure 2-15 below.)

FHP processes, unlike conventional processes, share the entire FHP address space. An FHP process' address space is not limited by artificial memory boundaries or addressing limitations. FHP processes can share all or part of their address space with other processes. Processes can share objects without requiring that multiple copies of an object be made.

tc96

Figure 2-15. FHP Processes' Address Spaces

### 2.2.3 Accelerated Addressing

We have seen that part of an FHP pointer is a 112-bit logical address (80-bit UID and 32-bit offset). This logical address locates a single bit in FHP's universal address space. However, most FHP systems will need to access only a small subset of FHP's 2**80 objects. Therefore, FHP systems can be designed to directly

reference a small set of "active" objects. Most FHP systems will manage this set of active objects with short, easily managed object identifiers. These system-dependent object identifiers are called Active Object Numbers. The non-architectural address acceleration technique that produces Active Object Numbers is called Logical Address Reduction. Logical Address Reduction is described further in Chapter 3.

### 2.2.4 Logical Input/Output

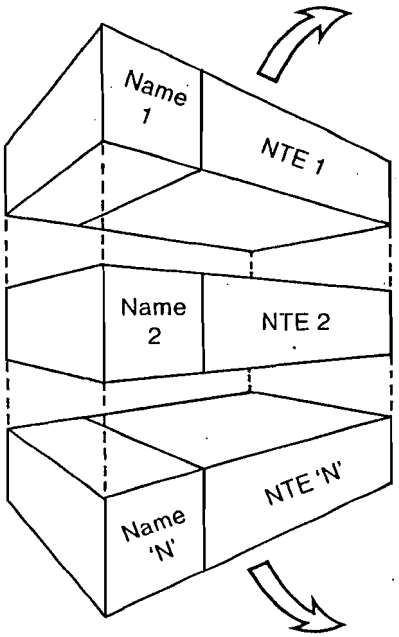
Most FHP systems will use separate hardware to perform Input/Output (I/O) operations in parallel with host machine operations. When a process requires I/O, a Physical I/O Directive (PIOD) is generated and sent to the I/O system's processor. The process making the request is then released while another process uses the available processor. When the I/O operation for the first process is complete, the process is again eligible to share processor resources.

The I/O system is responsible for receiving PIODs and converting them into Device I/O Directives (DIODs). DIODs are used to pass data both to and from an I/O device like a disk or tape drive. The high-level language programmer does not control these I/O functions himself; the generation of PIODs and their transformation to DIODs is controlled by the system.

6K

**Short Entry (64 bits)**

| Flags (12) | Type (4) | Base (16) | Length (16) | Displacement (16) |
|---|---|---|---|---|




**Long Entry (128 bits)**

| Flags (12) | Type (4) | Base (16) | Length (16) | Displacement 1 (16) |
|---|---|---|---|---|
| Displacement (16) | | Index (16) | Reserved (16) | IES (16) |

TC-00005-00

2-16

## 2.3 Namespace Addressing

Our explanation of S-languages indicated that an S-language interpreter (S-interpreter) is responsible for examining an S-language instruction's opcode and directing the host machine's operation. An S-language instruction's operands are referenced independently by Namespace Addressing. FHP's Namespace Addressing provides a flexible operand addressing mechanism that can be accelerated easily.

FHP systems use Namespace Addressing to automatically convert an S-language operand name into the operand's description or current value. These two functions are called operand resolution and operand evaluation. Resolving an operand name returns the operand's complete logical descriptor. Logical descriptors contain an operand's starting logical address (UID and offset), length, and type. Evaluating a name returns an operand's current value, length, and type.

### 2.3.1 Name Table Entries

An S-language operand's description is not carried in the S-language instruction. FHP systems' high-level language compilers automatically place descriptions of each item referenced by a procedure object into structures called Name Table Entries (NTEs). NTEs reside in the procedure's Name Table, which holds up to 2**16 entries. A Name Table is placed in a procedure object with the procedure's S-language instruction stream and header information. The procedure's Procedure Environment Descriptor (PED) has a Name Table Pointer (NTP) that locates its Name Table. A name taken from the S-language instruction stream indexes into the Name Table to locate the name's Name Table Entry.

tc5

Figure 2-16. Name Table and Name Table Entries

The FHP architecture defines two kinds of Name Table Entry. Short (64-bit) NTEs contain information needed to evaluate all operand names except names that represent arrays or data with a displacement into an object of more than 2**16 bits. Long NTEs contain information for referencing array elements or operands that are displaced more than 2**16 bits into an object.

The structure of the NTE allows addressing modes similar to those provided by conventional architectures, such as: direct and indirect, indexed, base, and base indexed. A short Name Table Entry contains the following fields:

* Flags
* Type
* Length
* Base
* Displacement

The long NTE contains all of the above information as well as index, inter-element spacing, and extended displacement fields. These Name Table Entry fields are described below.

The flags field indicates whether the NTE is short or long, and if long, whether the NTE represents an element of an array or a large-displacement operand. The flags field also indicates whether the length, base, or IES fields contain actual values, or names which must be evaluated. Resolving or evaluating an NTE can be recursive (which requires the resolution or evaluation of other names). For example, an NTE's length field can contain a name which is eventually evaluated to an integer value that represents an operand's length. The flags field also specifies base indirection. If base indirection is indicated, the name in the base field is evaluated and the result used as a pointer to another location.

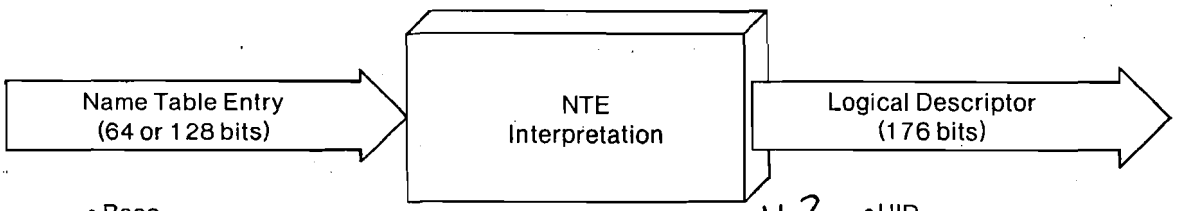
The non-architectural type field is available for use by S-language interpreters to indicate the operand's type (integer, floating point number, etc.). The last part of this field has operand alignment (justification) information.

The length field denotes the bit length of the operand value that will be read or written.

Namespace Addressing locates operands using base/displacement calculations. The base field designates one of three Architectural Base Registers (ABRs) that contain logical addresses (UID and offset). The specified ABR value is used as a base logical address when the NTE is interpreted. The base field can also contain a name which eventually references an ABR. The next section describes how ABRs are used in base/displacement calculations.

The displacement field is a signed literal value, which is added to the 32-bit offset of the NTE base value. Displacement is a 16-bit value in a short NTE and a 32-bit value using the long

OK




- Base
- Displacement
- Length
- Type
- Alignment
- Flags

112
32
16

- UID
- Offset
- Length
- Type
- Alignment
- Flags

TC-00028-00

2-17

TC28

NTE's displacement extension field.

The long Name Table Entry supplies three additional fields:

* Index
* Inter-element Spacing (IES)
* Extended displacement

The index and Inter-element Spacing fields are used to locate elements of arrays. The index field is a name whose evaluation yields the value of an element's subscript. For example, the 'I' part of the expression A(I) is A's index value. Namespace Addressing locates an element of a one-dimensional array (a vector) by interpreting two names. Namespace Addressing can reference elements of multi-dimensional arrays by interpreting two names for each dimension. A two-dimensional array such as B(J,K) requires that Namespace Addressing interpret four names. A three-dimensional array like C(I,J,K) requires the interpretation of six names. In Chapter 3, we show how Namespace Addressing locates an element of a one-dimensional array.

The Inter-element Spacing (IES) field specifies the difference between the starting addresses of two successive vector elements. The IES field can contain the name of an unsigned integer value, or the value itself. Namespace Addressing multiplies the value of the element's subscript by the value of the IES field to locate the proper array element.

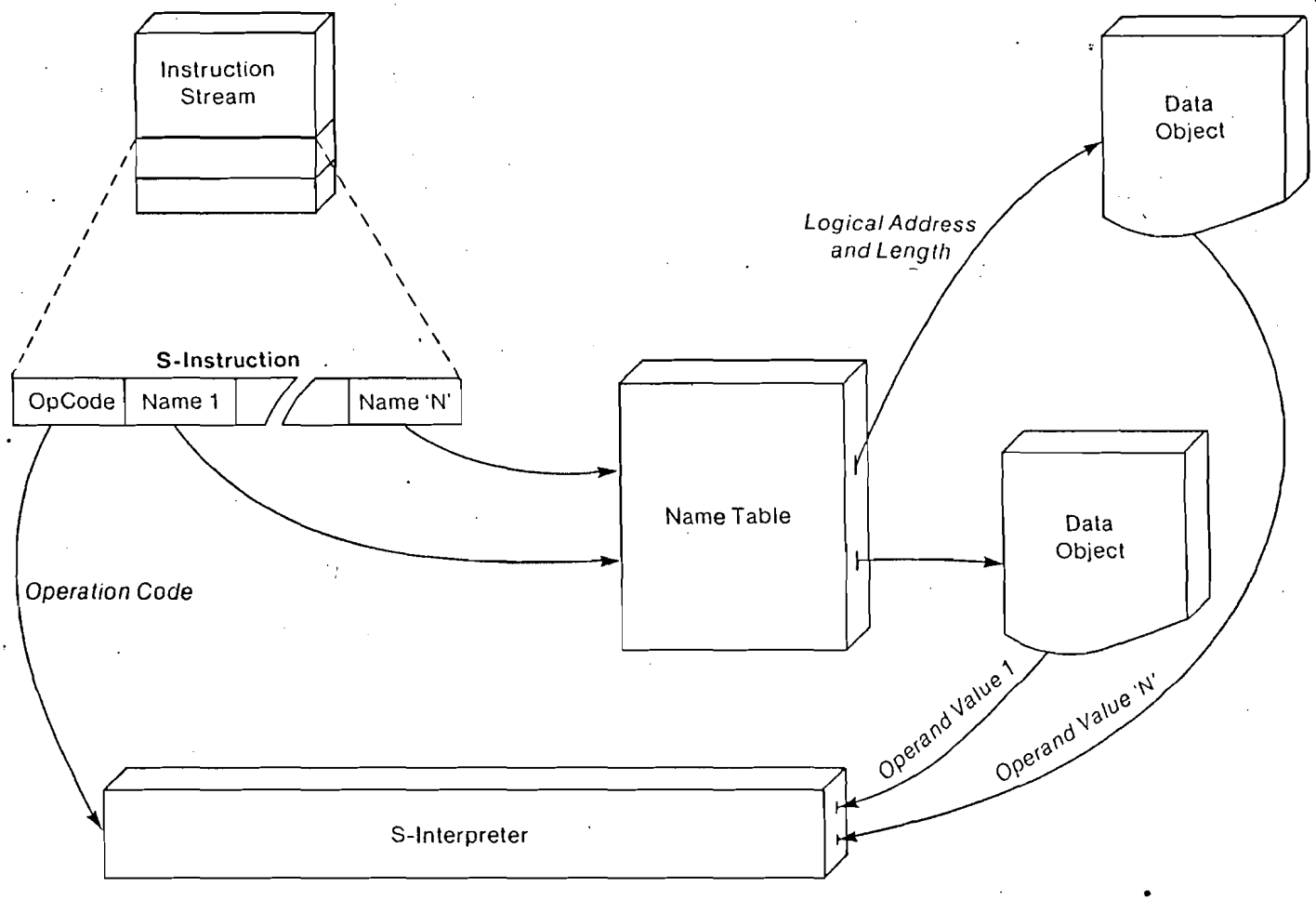
The third field in the long NTE is the 16-bit extended displacement field. This field is concatenated with the short NTE's displacement field to reference operands that are displaced more than 2**16 bits into an object.

## 2.3.2 Interpreting NTEs

An operand's NTE and Architectural Base Register information is used to create the operand's logical descriptor. A logical descriptor contains an operand's starting logical address (UID and offset), length, type, and alignment information.

tc28

Figure 2-17. Converting a Name Table Entry to a Logical Descriptor




TC-00018-00

2-18

CAPTION: Namespace Addressing uses S-instruction operand names as indexes into a procedure's Name Table. An operand name's Name Table Entry contains the information that Namespace Addressing uses to calculate the operand's logical address.

A logical address consists of a UID and an offset. The UID identifies the object that contains the operand; the offset specifies the operand's starting location within the object.

A system's S-language interpreter examines each S-language instruction's opcode independently of Namespace Addressing. The S-interpreter's microprogrammed routines direct the host machine as it performs the specified operation.

Namespace Addressing produces logical descriptors using base/displacement calculations. The base part of a calculation is taken from the value in one of three Architectural Base Registers (ABRs).

The three ABRs provide a process' "windows" to the FHP address space. Each of the ABRs contain a logical address that locates one of a process' data storage structures. The Procedure Base Pointer (PBP) locates the currently executing procedure object's code. The Static Data Pointer (SDP) locates a procedure's static data storage area. The Frame Pointer (FP) locates a procedure's arguments and automatic data. We discuss the Frame Pointer's use further in "Multi-programmed FHP Systems," later in this chapter.

ABRs are managed by an operating system, and are invisible to high-level language programmers. The operating system sets a process' ABR values when it creates the process. The ABR values are part of a process' state and are saved when a process is removed from a processor. The system automatically changes ABRs' contents as the result of operations like procedure calls and returns. To protect a process' integrity, high-level language programmers may not explicitly manipulate an ABR's contents.

The displacement part of the calculation determines the operand's location relative to its base ABR. Displacement is expressed as a signed literal value that is added to the offset portion of the base.

Figure 2-18 below illustrates how Namespace Addressing locates operands and returns their values to a processor.

tc18

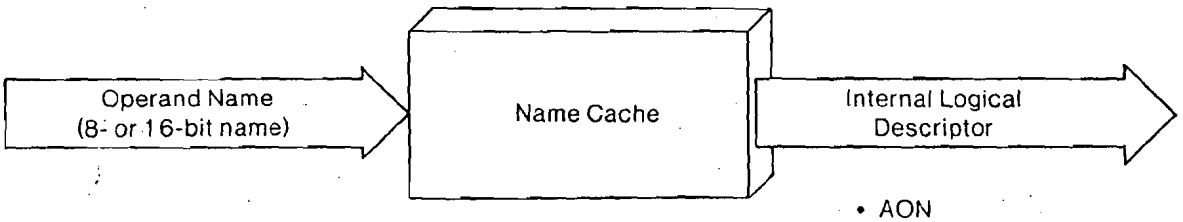
Figure 2-18. Locating Operands Using the Name Table

## 2.3.3 Branching

Namespace Addressing provides a branching facility that lets you execute S-language instructions out of sequence. Recall that each FHP process uses the 112-bit S-language Program Counter (PC). 80 bits of the PC contain the procedure object's UID. The system automatically increments the PC's 32-bit offset as S-language instruction stream syllables are fetched by a processor.

Information in S-language operand syllables can be used to modify the sequential flow through a procedure object's instructions. An S-language operation code (opcode) can designate

OK




- AON
- Offset
- Length
- Type
- Alignment
- Flags

TC-00049-00

2-19

8-bit or 16-bit name

8- or 16-bit name

an operand's contents as a branch that is relative to the PC. The PC relative branch value is taken directly from an operand syllable and added to the PC's offset value.

An opcode can also designate a branch location anywhere in the procedure (an absolute branch). The value of the absolute branch is obtained by evaluating an NTE to produce a 32-bit offset. This offset is added to the value in the Procedure Base Pointer ABR. The result replaces the S-language Program Counter's offset value.

## Accelerated Name Table Entry Interpretation

Most models of the FHP architecture will use a non-architectural Name Cache to accelerate NTE interpretation and directly convert operand names to logical descriptors. The performance advantage of a Name Cache is discussed further in Chapter 3.

tc49

Figure 2-19. Name Cache Acceleration

## 2.4 Bit Granularity

Conventional architectures are usually restricted to addressing schemes based on word size. Some architectures use 16-bit words, others use 32-bit words, and so on. Computers with a relatively large word size like 32 bits can perform powerful operations during each instruction cycle. Popular "supermini" and "mainframe" computers generally use word lengths of 32 bits or more.

Word-based addressing does not always fulfill the requirements of high-level languages or the data they process. For example, fixed word-length systems do not deal effectively with boolean (bit) manipulations. An architecture need not be restricted to fixed, word-length manipulations. It can, in fact, be more efficient by letting the source language define word lengths. FHP uses bit-granular addressing to eliminate the restrictions of fixed word-length systems.

### FHP's Bit-Granular Addressing

Bit granularity is the most adaptable form of addressing, because it lets you define and manipulate structures of any length. Since bit granularity can deal primitively with arbitrary bit fields, it lets FHP process high-level languages like SPL naturally and efficiently.

The FHP architecture has no notion of a "word" in the traditional sense of a fixed-length, addressable unit. To compare an FHP system's word size to that of a conventional architecture, you can look at FHP system's internal data path width. For example, the FHP SPRINT has 32-bit internal data paths. In FHP, the smallest addressable unit is a single bit, and the largest is a complete object of (2**32)-1 bits. FHP's bit-granular architecture will match the requirements of a wide range of existing or future high-level languages.

## 2.5 FHP's Security System

Computer security is becoming very important to system users, but conventional architectures do little to support comprehensive security measures. Security should include protection of user data, system data, and the physical facility.

These are some design considerations for a well-protected architecture:

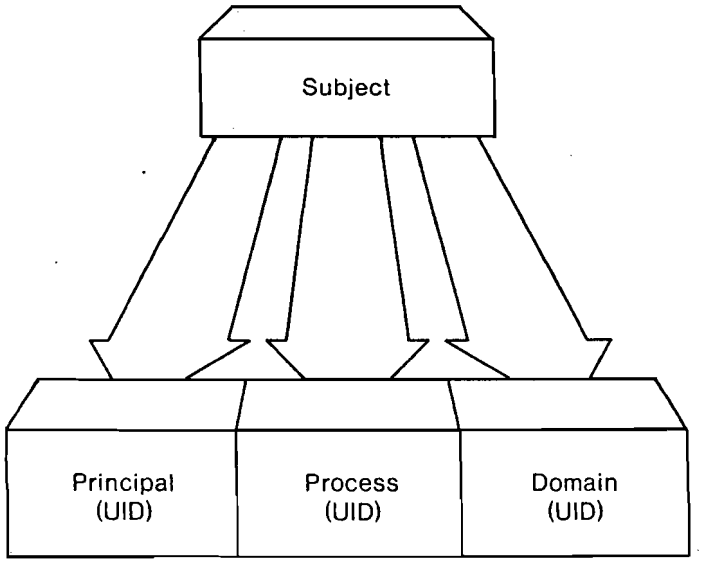
* Design security measures so they can be easily implemented in hardware. This will allow efficient performance;
* Let access to data be controlled by its owner.
* Let "no access" be the protection mechanism's default condition.
* Keep the protection mechanism flexible but simple. Elaborate protection mechanisms are difficult to design, verify, and implement. Simple protection methods usually have fewer loopholes than more elaborate approaches. Simple methods also require less system overhead.

FHP is a secure architecture that controls what rights a user, called a subject, has to an object's primitive or extended operations. Recall that all primitive objects provide data and non-data-mode operations. Data-mode operations are read, write, and execute; non-data-mode operations let you access and manipulate an object's attributes. A subject's rights to perform these operations are contained in an object's Access Control List (ACL).

In the next three sections, we discuss how FHP systems control a subject's access rights to primitive and Extended-type Objects.

### 2.5.1 Subjects

An FHP process is always associated with, or "bound," to a subject. The process performs work only for that subject. An FHP subject contains more information than just a user's name. Subjects have three components whose contents are compared to the three subject components in an object's ACL entry. One component is reserved for future use. A subject's three architecturally defined components are:




TC-00031-00

* who the process performs for, the principal;

* the identity of the process bound to the subject; and

* where the process performs, a domain.

The system assigns 80-bit UIDs to each of these components. It assigns null UIDs (all zeros) to components which can match any ACL principal, process, or domain.

tc31

Figure 2-20. An FHP Subject's Three Components

The principal is the name of the user who "owns" the process. A principal can represent a person's name, the name of a group, or all users of the system.

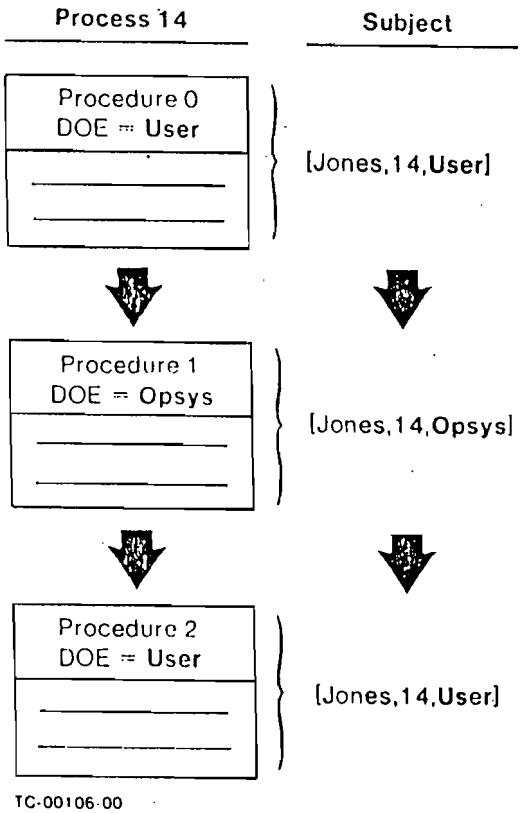
The operating system controls programs by running them in processes. A subject's process component value (a UID) is set by the operating system. System programmers can use a subject's process component to let only certain processes access an object. For example, if you set an ACL entry's process value to 14, then a subject must be bound to process 14 to reference the object.

Domains add another powerful dimension to FHP's security mechanism by assigning access rights to procedures. Domains let you control what subject is bound to a process, as each procedure is run. Domains let you create protective "walls" around groups of procedure objects.

One of a procedure object's attributes is its Domain of Execution (DOE). A procedure with no specific DOE can execute in any domain. If you specify a DOE for a procedure, that procedure can be called from any domain, but can execute only in the specified DOE.

A subject's principal and process values can not change during the life of a process. A subject's domain value automatically changes when the process executes a procedure in another domain (see Figure 2-21). The subject's domain value automatically returns to its original state when execution in the target procedure completes. This feature lets programmers give a process temporary access rights to a procedure. Therefore, systems need not expend resources to create another process when a user process requires operating system assistance. The system continues to execute the original process, but the process' access rights are

ok




2-21

CAPTION: An FHP system can change a subject's access privileges by temporarily changing the subject's domain component.

changed when it runs in another domain.

tc106

Figure 2-21. Process Executing Procedures From Different Domains

An FHP system can have any number of domains. For example, the SPRINT E/110 has three domains: two used by the operating system and one by system users.

Comparing Subjects and ACL Entries

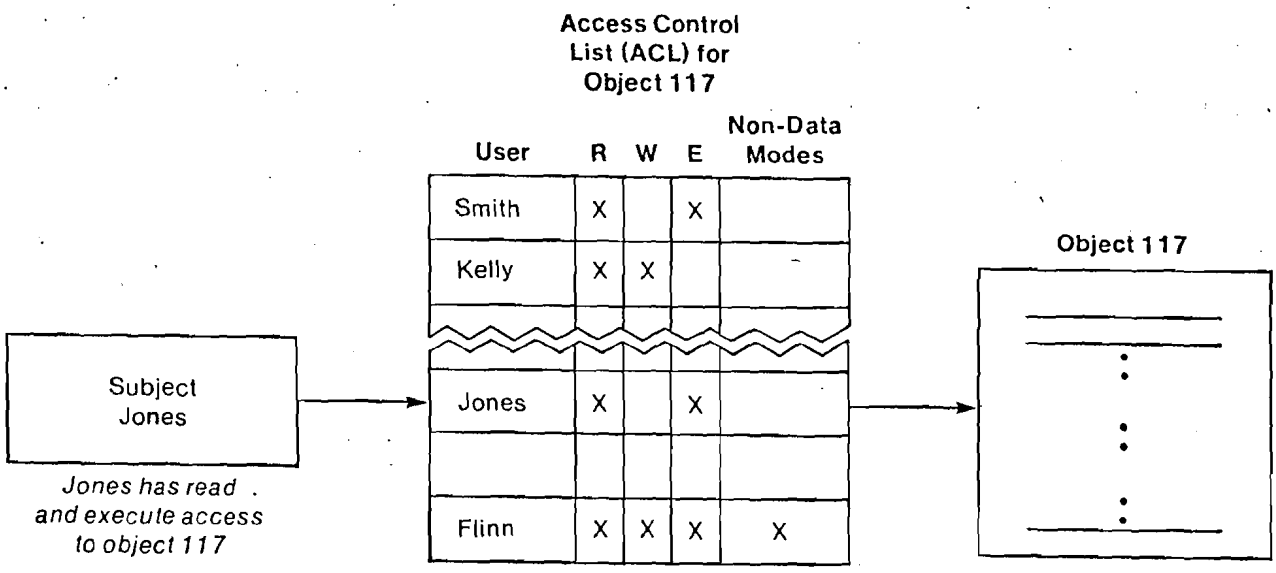
A subject's ACL entry values need not match a calling subject exactly. The three values in a subject's ACL entry can be specific or general. This lets programmers "fine-tune" which subject or group of subjects has access to an object. An example of an general subject entry is [*,*,Opsys], which matches any principal and any process in the operating system (Opsys) domain. The star (*) is a template that matches any component value. A subject entry such as [Ash,11,User] is specific; it matches only the principal Ash and process 11 running in the user domain.

## 2.5.2 Protecting Primitive Objects

Every FHP object has a primitive Access Control List (ACL). An ACL contains a list of entries with the name of every subject who can reference the object, and indicates which primitive operations each subject can perform. The subject/ACL relationship lets programmers deny attempts to execute a data object, write into a procedure object, or change an object's attributes.

For example, the ACL entry [Jones,*,*][R,E], means that Jones has the right to read (R) and execute (E) an object, but may not write (W) into the object or change its attributes (see Figure 2-22 below). The ACL entry, [*,*,*][R,W], would mean that all subjects (represented by '*') can read and write an object.

In Figure 2-22 below, Flinn can read, write and execute object 117. Flinn also has non-data mode access rights to the object. A subject with non-data mode access to an object is considered its owner. An object's owner can delete the object, or change its attributes and ACL.




TC-00020-00 2-22

CAPTION: All primitive objects can be protected with an Access Control List. This list contains the access rights for each subject that can reference the object.

tc20

Figure 2-22. Protecting a Primitive Object

## 2.5.3 Protecting Extended-type Objects

We discussed Extended-type Objects and Extended-type Managers (ETMs) earlier in this chapter. The FHP architecture has flexible mechanisms to control access to ETMs and their associated Extended-type Objects. FHP's extended protection mechanism is an enhanced version of the subject/ACL relationship that we discussed above.

Extended-type Objects can have Extended Access Control Lists (EACLs) in addition to their primitive ACLs. An EACL contains entries that describe a subject's rights to perform extended operations. A subject's right to perform an extended operation is verified in three steps:

1) The operating system checks the ETM's ACL to ensure that the subject can execute the ETM's procedures.

2) An ETM procedure examines the extended object's EACL to verify that the subject can perform the requested operation.

3) The operating system checks the Extended-type Object's ACL to ensure that the subject has the proper primitive access rights to the Extended-type Object.

There are many ways of establishing a protection scheme with this three-step procedure. We illustrate one way with the following example. Our example shows a typical method of protecting a group of Extended-type Objects.

### Example of Extended Protection

Please refer to Figure 2-23 below as you read this section. Assume that a programmer has created an Extended-type Object Manager (ETM) that manages a group of stacks. This ETM (UID 38846) has four procedures: creating a stack, deleting a stack, pushing data onto a stack, and popping data from a stack. Recall that you can store data by pushing it on the stack. You retrieve this data by popping it from the stack.

For simplicity, we show two stacks: one dedicated to the subject Jones, and one dedicated to subject Smith. We want to let Jones and Smith perform push and pop operations on their respective stacks only; we do not want them to create or delete stacks. The only subject who can create or delete stacks is Lewis, who has more authority than Jones or Smith.

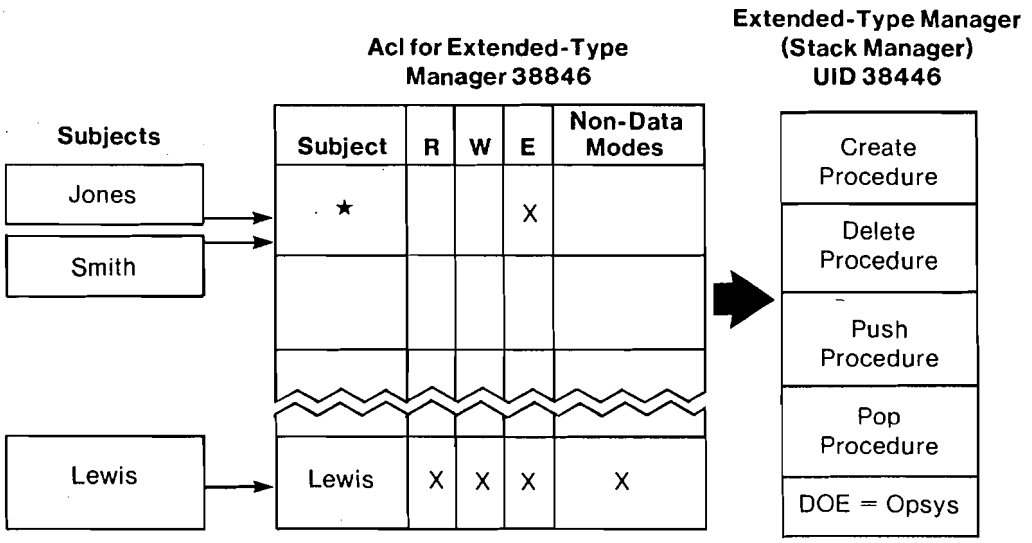
The figure below shows kinds of access control list: the ETM's ACL, and the Extended-type Object's EACLs and ACLs. What do these access control lists tell us? The ETM's ACL indicates that any subject (*) can execute the four procedures in the ETM (UID 38846). Lewis subject can execute and modify the procedures in the ETM, as well as change the ETM's attributes.

Notice that the ETM's Domain of Execution (DOE) is set to the operating system domain (Opsys). This means that a process executing any procedure in this ETM has its subject's domain component value changed to Upsys.

Each stack object has its own EACL. The ETM's procedures are responsible for examining an EACL to determine a subject's access rights. The EACL for Jones' stack indicates that Jones can perform push and pop operations on the stack. Lewis can perform all four operations: push, pop, create, and delete. No other subjects can read or manipulate the data in this stack. The same is true for Smith's stack. Smith can perform the extended operations push and pop, while Lewis can perform all four operations. No other subject can access Smith's stack.

The stack objects are further protected by their primitive ACLs. In this example, these ACLs allow read and write access to the stack objects from the operating system domain only. Therefore, processes running in other domains cannot manipulate these stacks. The pop operation requires read access to the Extended-type Object, the push operation requires write access.

**STEP 1**




**STEP 2**

**Extended Access Control Lists for Extended-Type Objects of Type "Stack"**

| Subject | Pop | Push | Create | Delete |
|---|---|---|---|---|
| Jones | X | X | | |
| Lewis | X | X | X | X |

EACL for Object 38820
(Jones' stack)

| Subject | Pop | Push | Create | Delete |
|---|---|---|---|---|
| Smith | X | X | | |
| Lewis | X | X | X | X |

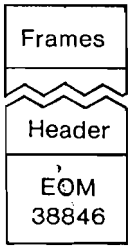
EACL for Object 19926
(Smith's stack)

**STEP 3**

**Jones' Stack UID 38820**

| Subject | R | W | E |
|---|---|---|---|
| *,*,Opsys | X | X | |

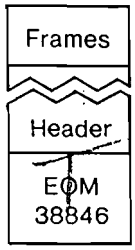
ACL for Object 38820




**Smith's Stack UID 19926**

| Subject | R | W | E |
|---|---|---|---|
| *,*,Opsys | X | X | |

ACL for Object 19926




TC-00021-00

tc21

Figure 2-23. Protecting Extended-type Objects

## 2.5.4 Bounds Checking

FHP's security system tests each reference to an object to make sure that the reference stays within the bounds of the object. One of an object's attributes is its size, expressed in bits. A protection error is signalled if you try to read data past an object's maximum length. An error is also signalled if you try to write more data into an object than it can hold. The system can increase the size of the object when an extent error occurs.

## 2.5.5 Architectural Call

One of the most important parts of FHP's security system is provided by a system function called Architectural_Call. Architectural Call guarantees secure inter and intra-procedure object calls. FHP S-languages have similar Architectural Call instructions which are issued when one procedure calls another. The high-level language programmer does not explicitly issue Architectural Calls; a call from a high-level language implies that an Architectural Call will be performed.

Architectural Call instructions use a pointer to specify the call's target procedure. The call instruction also specifies the number of arguments it is passing and identifies each argument by name.

An Architectural Call has three basic steps, followed by a return function. The next section describes each step. Architectural Call:

1) saves the calling procedure's environmental information;

2) validates the calling subject's rights to the target procedure; and

3) prepares the target procedure's environment.

First, information about the calling procedure is saved so that it can continue after control returns from the target procedure. Second, the target procedure's Domain of Execution (DOE) is compared to the calling subject's domain value, and the target procedure's ACL is compared to the calling subject. A

cross-domain Architectural Call is performed if the target procedure has a DOE different from the calling subject's domain value. A cross-domain call automatically changes the calling subject's to the DOE of the target procedure. On return, the subject's domain value is returned to its pre-call setting.

Architectural Call also tests to make sure that the calling procedure has the proper access rights to any arguments that it tries to pass to the target procedure. If the Architectural Call fails because the subject does not have proper access rights to arguments or the target procedure, an error is signalled, and the call mechanism suspends the reference. The calling process will not disturb any procedure in any other domain.

Architectural Call's third step is to examine the target procedure's Procedure Environment Descriptor, prepare the procedure's environment, and begin executing its S-instructions.

## An Example of Architectural Call

A process running on an FHP system will usually encounter a number of programmer calls for operating system assistance. If we assume that this process is executing in a user domain, then system calls are made to procedures that reside in an operating system domain.

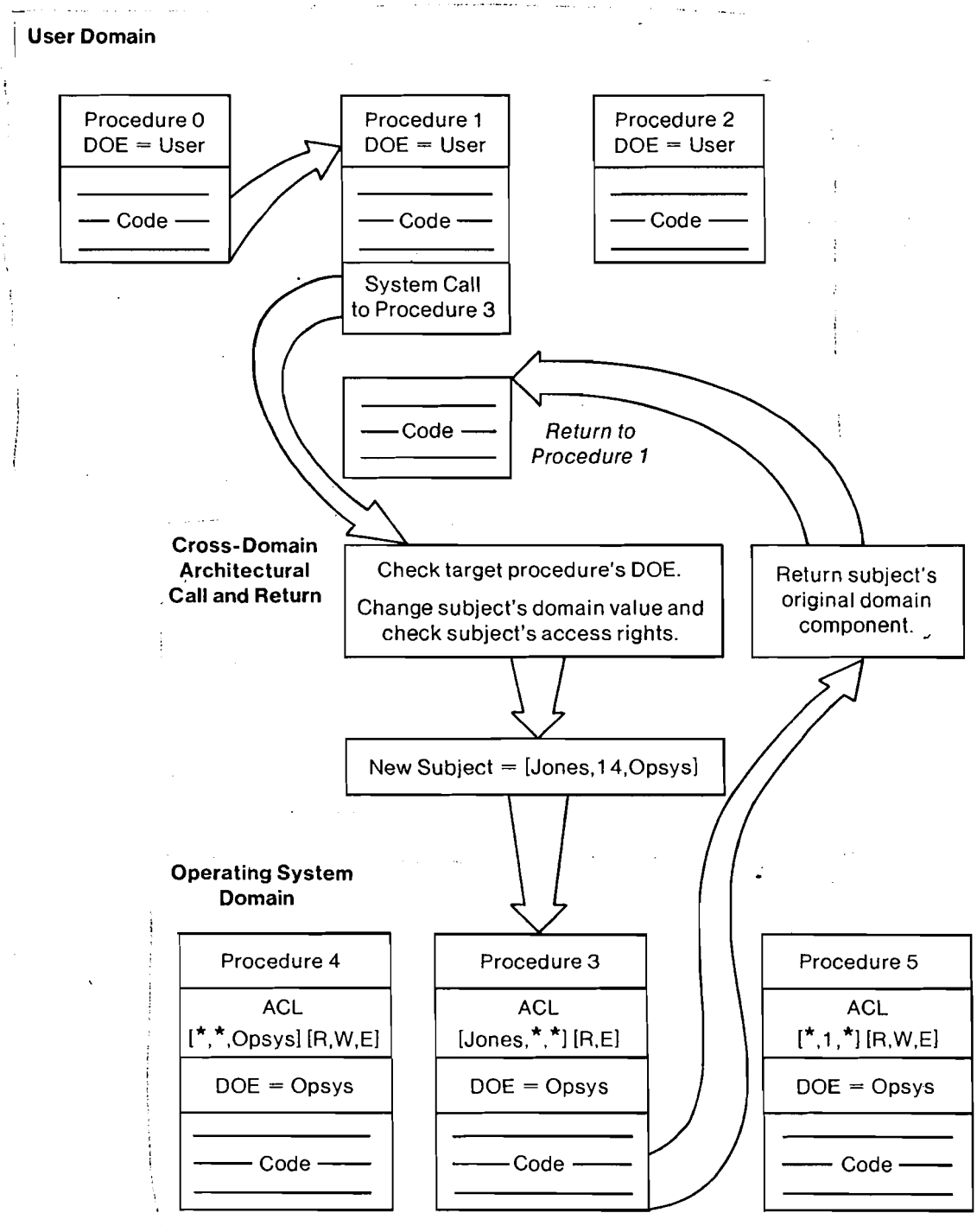
In the Architectural Call example shown in Figure 2-24 below, Procedures 0, 1, and 2 are in the user domain. Procedures 3, 4, and 5 are in the operating system domain. Notice that Procedure 4 can not be called directly from any procedure in the user domain, because its ACL entry's domain value is Opsys. Also, Procedure 5 may not be called from any procedure that is not being run by Process 1, because its ACL entry's process component is 1.

Let's assume that Process 14 runs Procedure 0 to completion, then begins executing Procedure 1. This procedure then issues a call to Procedure 3. Procedure 3's DOE indicates that it is in the operating system domain. One of Procedure 3's ACL entries is [Jones,*,*][R,E]. This entry lets any subject with a Jones principal component read or execute the object.

Architectural Call changes Process 14's domain component from the user domain to the operating system domain. Architectural Call now checks the new subject's rights to access the target procedure by examining the target procedure's ACL. This access right check shows that Process 14 can call and execute Procedure 4, because the new subject matches the subject template in the ACL entry. Note that Procedure 5 still cannot be called from Process 14, because the procedure's ACL specifies a process component of 1.

**Initial Subject Bound to Process 14**

| Principal | Process | Domain |
|---|---|---|
| Jones | 14 | User |




TC-00097-00

2-21

When Procedure 3 has completed, Architectural Call automatically returns the subject's domain component to "user," which restores the subject's original access privileges.

tc97

Figure 2-24. Architectural Call **See attached drawing**

### 2.5.6 Accelerated Protection Checking

Checking a subject's access rights to an FHP object can be accelerated with system tables and caches. Most FHP systems will perform protection checking by comparing an active subject to an active object. This information would be stored in a table called the Active Primitive Access Matrix (APAM).

Chapter 3 describes how the APAM and a protection cache can be used to accelerate protection checking.

## 2.6 Multi-programmed FHP Systems

Up to now, we have described how an FHP system runs a single process. The FHP architecture provides multi-programming features to support multiple users. Multi-programming lets multiple processes use Virtual Processors to share the resources of a single host machine. Multi-programmed systems use system resources efficiently; if one process were waiting for something like I/O service, another process could use the idle host machine. When the first process' I/O is complete, the process can again contend for host machine resources.

This section introduces some FHP features that support multiple user applications. Multiple user scheduling is supported by an architecturally defined Kernel operating system. We describe how processes and processors are scheduled, and how a system saves a suspended process' state. We also describe how FHP systems use Virtual Processors (VPs) to support multiple user applications.

### 2.6.1 Process-to-processor Scheduling

In multi-programmed applications, host machine sharing is controlled by system software. The system automatically allocates processor resources with model-dependent scheduling algorithms.

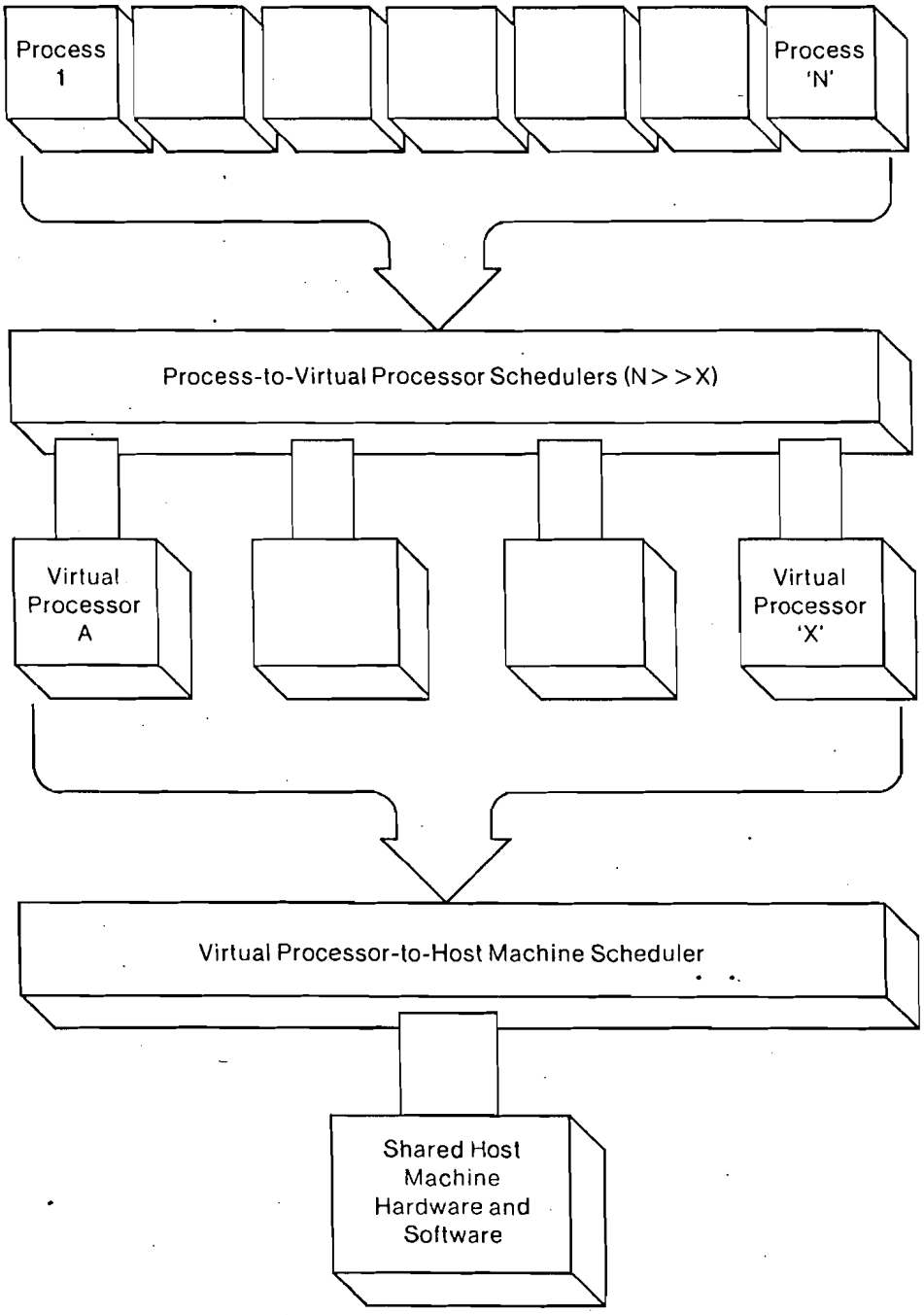
Figure 2-25 below shows how multiple processes share an FHP system's host machine. Notice the two levels of scheduling: the higher level allocates processes to Virtual Processors; the lower level shares the host machine among the Virtual Processors. An FHP process' architectural contract is with a Virtual Processor. FHP systems use Virtual Processors to efficiently manage valuable system resources such as memory and CPU time.

A Virtual Processor provides a richer and more powerful interface than that provided by a host machine's hardware and firmware alone. An FHP Virtual Processor uses a combination of software and firmware to enhance the user-visible interface to a system's host machine. Virtual Processors (VPs) provide an interface between a system's host machine and FHP processes.

Different models of the architecture can accelerate selected VP functions into the host machine without disturbing the the process-to-VP interface. A process is not aware of this acceleration; the process simply runs faster.

A process need not run to completion on a specific Virtual Processor, because all VPs sharing a host machine present the same appearance to a process. A process may begin running on VP 'A', be pre-empted, and finish running on VP 'B'. Process, VP, and host machine management is provided by operating system software, and is

CAPTION: All processes on an FHP system compete for system resources. These resources are provided by a set of Virtual Processors, which share the host-machine's hardware and software. There are usually more processes than Virtual Processors, so the architecture provides two schedulers. One scheduler shares a system's Virtual Processors among its processes. The other shares the host machine among the Virtual Processors.




TC-00009-00

invisible to high-level language programmers.

The system manager establishes the number of VPs in a system. The number of VPs is based on the system's user load and the resources available to the system, such as the amount of main memory.

tc9

Figure 2-25. Scheduling Multiple FHP Processes

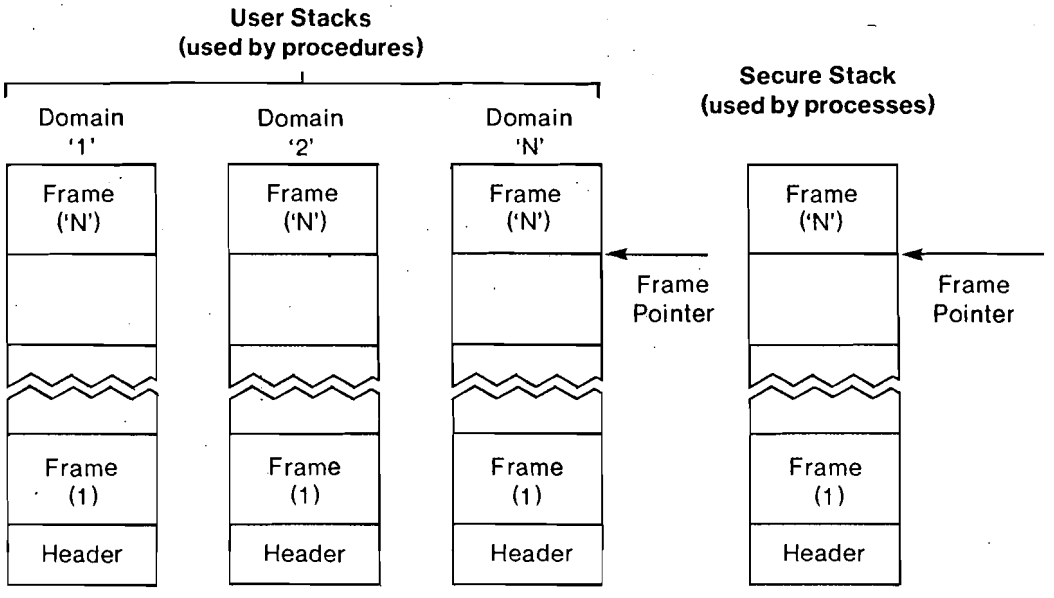
Multiple-user FHP systems normally have more processes than VPs, so that processes compete for Virtual Processors. In Figure 2-24 above, the value of 'n' is greater than value of 'X'. Processes compete for Virtual Processor service based on the process' relative priorities. Therefore, FHP systems must be able to save information about a process' environment so that the process can be restarted if it is suspended. This information is called the process' state. A process' state is a snapshot of its environment at the instant it is suspended (removed from a processor).

An FHP process' state has two parts: macro-state and micro-state. Macro-state includes information about a process' currently active procedure. Macro-state includes: the values in the currently active procedure's Architectural Base Registers; the offset portion of the S-language Program Counter; the location of the procedure's current activation record; and the location of the procedure's Entry Descriptor.

Micro-state includes information about a Virtual Processor and its supporting host machine. This includes the value in the host machine's micro-program counter, identity of the process' subject, and other machine-dependent data structures. FHP processes use dedicated stack objects to allocate a procedure's storage space, and save process state. Macro-state and micro-state can not be modified by a target procedure.

## 2.6.2 A Process' Stacks

Each FHP process has two types of stacks: architecturally defined user stacks, and a model-dependent per-process secure stack (See Figure 2-26 below). User stacks maintain procedures' activation records. An activation record contains data that a procedure can modify each time the system runs (activates) the procedure. The record also contains and pointers to any arguments the procedure uses. Each procedure activation has access to its own, independent activation record in a user stack.




TC-00098-00

2-26

The secure stack is used as a convenient means of saving a process' macro-state and micro-state. Storing this information on the secure stack guarantees that Architectural Calls do not compromise a system's security. Target procedures cannot modify a calling procedure's state information.

These stacks are implemented as extended-type objects of type "user stack" and "secure stack." Both stacks have stack headers that provide information about the stack's type and size. Data in the stacks is contained in stack frames. The system locates a stack frame using the stack's frame pointer, which is a 32-bit offset into the stack object. Frames are "pushed" onto the top of a stack by moving the stack's frame pointer away from the stack header; frames are "popped" from the stack by moving the frame pointer toward the stack header.

tc98

Figure 2-26. A Process' Stacks

## User Stacks

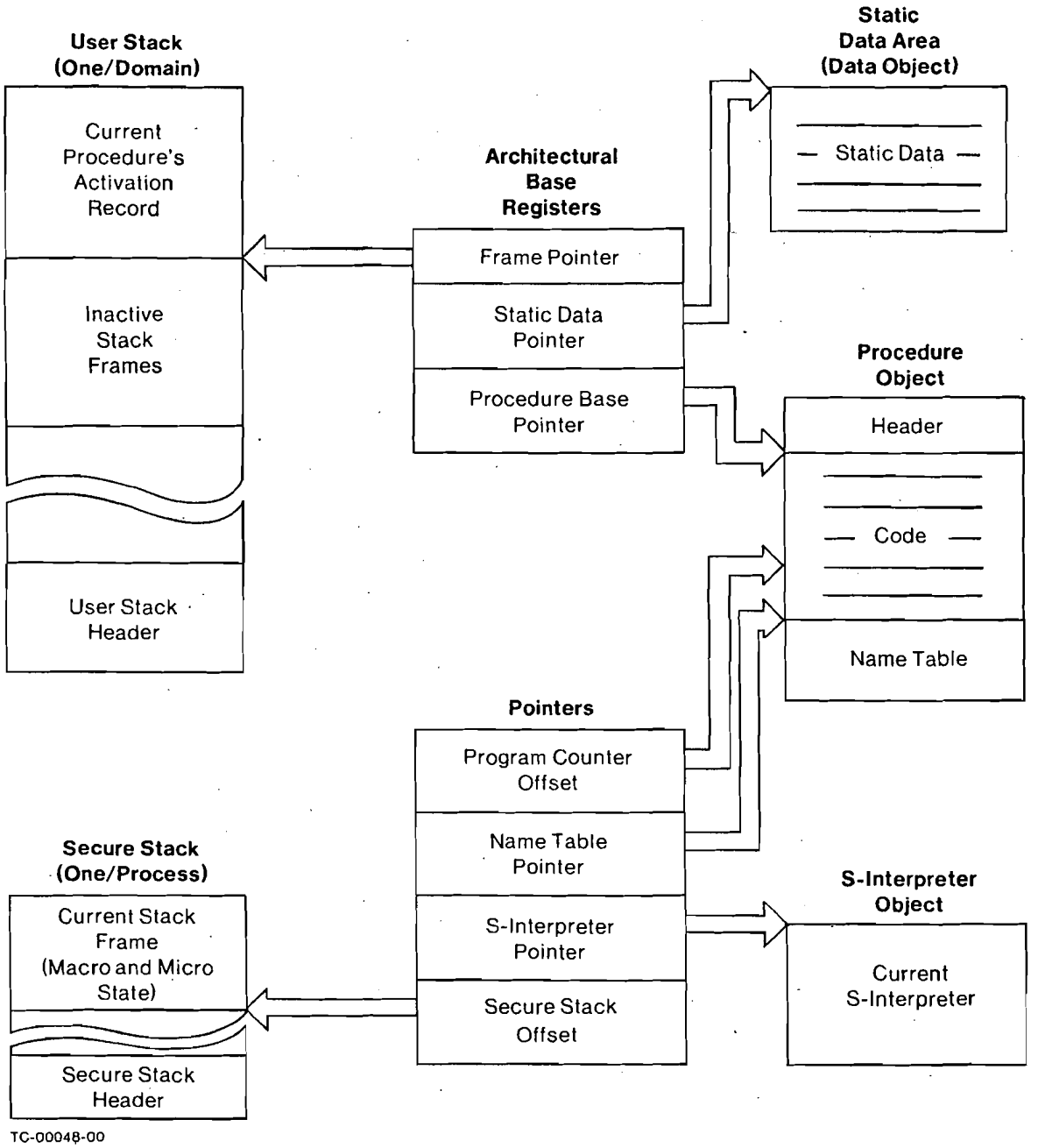
A process has one user stack for each domain that the process runs in. For example, a process in a system with three domains has three user stacks. Processes automatically switch user stacks when a procedure in the process initiates a cross-domain call.

User stacks contain non-secure information for procedure calls. The stack also contains pointers to any arguments that a procedure is passing in the call.

User stacks also contain data that a procedure modifies each time the procedure is activated. This is the procedure's automatic data. Automatic data includes variables that a procedure uses for temporary storage. Automatic data storage locations are generated each time a procedure is activated. Saving a procedure's activation records in a user stack supports re-entrant S-language procedures. This means that more than one process can execute FHP procedures concurrently. Each activation of a procedure within a domain has its own automatic storage area in a user stack frame.

## Secure Stack

Each FHP process has one secure stack. Information in the secure stack is accessed by operating system procedures only.




TC-00048-00

2-27

The secure stack guarantees that Architectural Calls do not compromise a system's security. Procedure-to-procedure linkage information is placed on this stack, so that proper returns are guaranteed. During an Architectural Call, macro-state and micro-state state information is collected and pushed onto the secure stack's top frame. This frame is removed from the stack when the call is complete, so that a Virtual Processor and the host machine can be returned to their pre-call state.

Static data, which is common to all activations of a procedure, is kept in a separate static data area. This area, contained in a data object, is located by the procedure's Static Data Pointer ABR. The logical address in the Static Data Pointer ABR is saved as part of a process' macro-state.

## Summarizing A Process' Environment

A simplified view of an FHP process' environment is summarized in Figure 2-27 below. This illustration shows a process' storage areas, stacks, and pointers to important data structures.

tc48

Figure 2-27. A Process' Environment

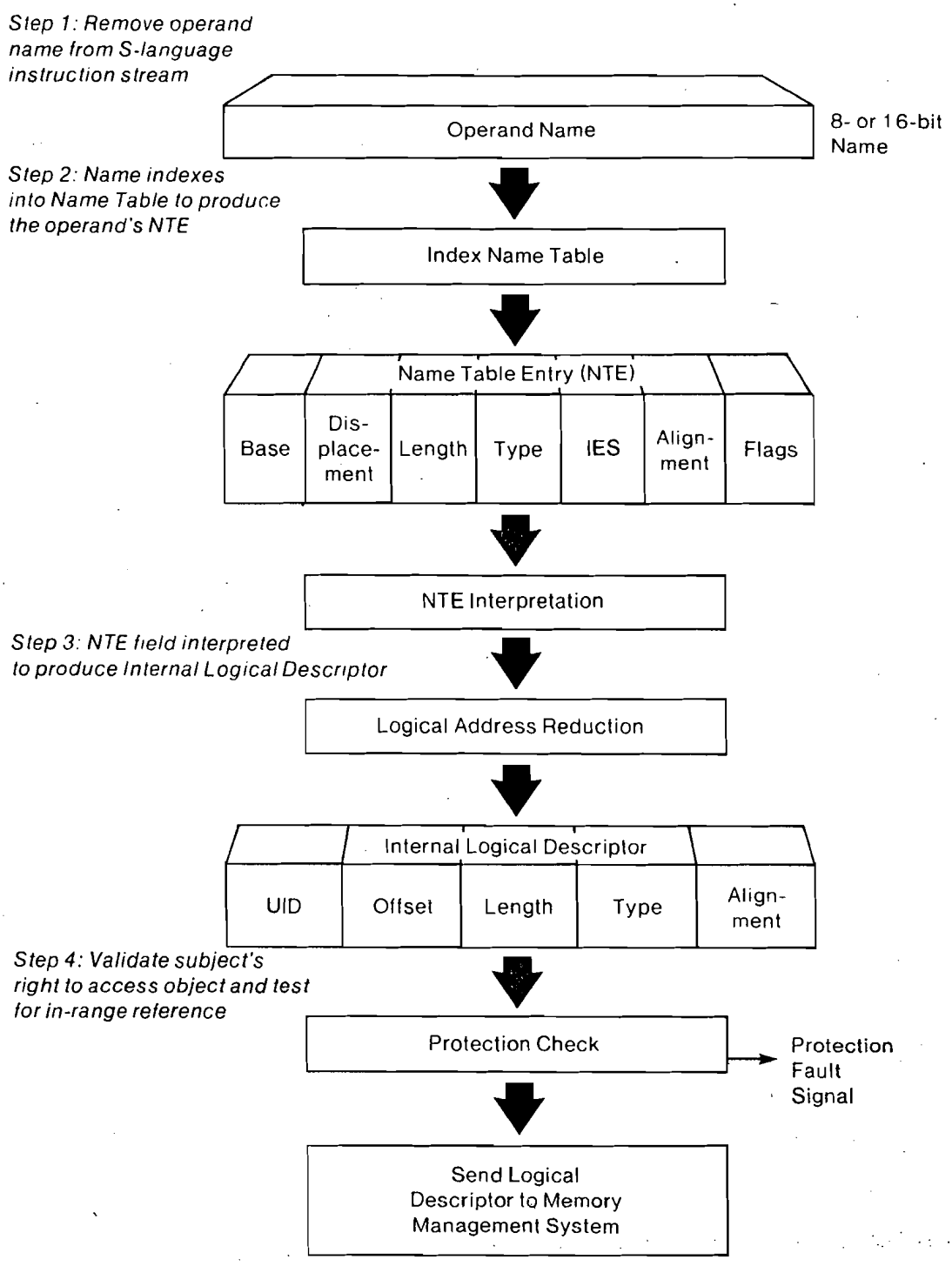
## 2.7 Summary

This chapter has outlined many of the FHP architecture's major features, including:

* S-languages and dynamically switched S-language interpreters;
* FHP's universal memory, which contains uniquely named objects;
* Namespace Addressing, used to address FHP's memory; and
* FHP's powerful security system.

The next chapter shows you how these features are integrated into the architecture, and gives examples of unaccelerated and accelerated FHP addressing. The chapter also shows how FHP systems convert logical descriptors into main memory addresses.

--End of Chapter--


Step 1: Remove operand name from S-language instruction stream
Operand Name
8- or 16-bit Name
Step 2: Name indexes into Name Table to produce the operand's NTE
Index Name Table
Name Table Entry (NTE)
Base
Dis-place-ment
Length
Type
IES
Align-ment
Flags
NTE Interpretation
Step 3: NTE field interpreted to produce Internal Logical Descriptor
Logical Address Reduction
Internal Logical Descriptor
UID
Offset
Length
Type
Align-ment
Step 4: Validate subject's right to access object and test for in-range reference
Protection Check
Protection Fault Signal
Send Logical Descriptor to Memory Management System

# Chapter 3
# FHP System Addressing Summary

In the previous chapter, we introduced the FHP architecture's individual features and discussed their contribution to efficient processing of high-level language programs. This chapter shows how these features are integrated and gives examples of unaccelerated and accelerated addressing. The chapter:

1) summarizes FHP addressing and protection checking;

2) describes how typical FHP systems can apply basic acceleration techniques to addressing and protection checking;

3) describes how typical FHP systems address data in main (physical) memory; and

4) demonstrates how FHP systems can use inexpensive semiconductor memories to further accelerate addressing and protection checking.

## 3.1 Architectural Addressing Summary

This section presents the steps that an unaccelerated FHP system uses to transform an S-language operand name into the operand's logical address. We use two examples to demonstrate FHP's addressing and protection checking functions. The first example shows you how an FHP system references a scalar operand. This example uses one of the simple FORTRAN S-language instructions introduced in Chapter 2. The second example describes how FHP systems address a more complex structure: an element of a single dimension array. Both examples contain simplified illustrations of the data structures used by FHP's addressing and protection checking mechanisms.

Figure 3-1 below illustrates how an unaccelerated FHP system creates an operand's logical address. This figure applies to both addressing examples presented in this section.

tc121

Figure 3-1. Architectural Addressing Summary

| Operation Code | Name for Variable 'L' | Name for Variable 'I' |
|---|---|---|
| IADD2 | 123 | 124 |

TC-00037-00

3-2

### 3.1.1 Referencing a Scalar Operand

For our first addressing example, assume that an FHP process is running a procedure for the user Jones. Let's say that the procedure's instruction stream contains the FORTRAN S-language statement IADD2 L,I that we used Chapter 2 This statement requires that the system return the value of the variable 'L', so that the processor's arithmetic unit can add this value to 'I', and place the result into 'I'. We'll assume that the 'L' variable contains a 16-bit integer value.

The following example describes how FHP systems use Namespace Addressing to locate the value of the variable 'L', and how protection checking is performed.

#### 3.1.1.1 Remove Operand Name From Instruction Stream

The first step (Figure 3-2) is to remove the operand's numerical name from the procedure object's instruction stream. A FORTRAN compiler creates the operand's name and corresponding Name Table Entry (NTE) and places the NTE into the procedure's Name Table. We'll assume that the numerical name that the compiler assigned to the variable 'L' is 123.

tc37

Figure 3-2. Names in the S-instruction

The information required to remove the operand is the operand length 'k', which is found in the procedure's Procedure Environment Descriptor (PED). 'k' will be either 8 or 16 bits; for this example, assume that 'k' is 8 bits. The currently executing S-language instruction is located by the S-language Program Counter (PC). The PC always points to the beginning of an instruction's operation code. The PC is automatically incremented by the number of bits in an instruction when the processor executes the instruction.

#### 3.1.1.2 Index Into Name Table to Produce NTE

The name (123) of the variable 'L' is used as an index into the procedure's Name Table. Namespace Addressing uses the Name Table Pointer (NTP) to locate the proper Name Table. The NTP is contained in the procedure's Procedure Environment Descriptor.

Using the name to index the Name Table produces the names' Name Table Entry (NTE). The NTE contains information that Names-

| Flags and Alignment | Type | Base | Length | Displacement |
|---|---|---|---|---|
| • Short NTE<br>• Right-justified<br>• Base is an ABR | Integer | Frame Pointer | 16 bits | 16048 bits |

TC-00038-00

pace Addressing interprets to create the operand's description; in our case, information describing the 16-bit integer variable 'L'. L's NTE expresses the operand's location in terms of a base and a displacement from the base. The base part of the calculation resolves to one of three currently defined Architectural Base Registers (ABRs). We say "resolves" to an ABR because the base field in the Name Table Entry can contain a name that Namespace Addressing must convert to an ABR. The displacement part of the calculation locates the operand's starting location relative to the specified ABR value.

The other fields in the NTE specify the operand's length, type, and alignment (justification). The flags field specifies how the other fields in the NTE will be interpreted. The flags field in our example NTE indicates that the operand is described by a short NTE, is right-justified, and that the base field represents an ABR.

The long NTE would be used if our example operand were longer than 16 bits, or an element of an array. The long NTE has fields with the array's index and Inter-element Spacing (IES), used to calculate an element's location. We give you an example of how Namespace Addressing references an element in an array later in this chapter.

This example NTE is shown in Figure 3-3 below.

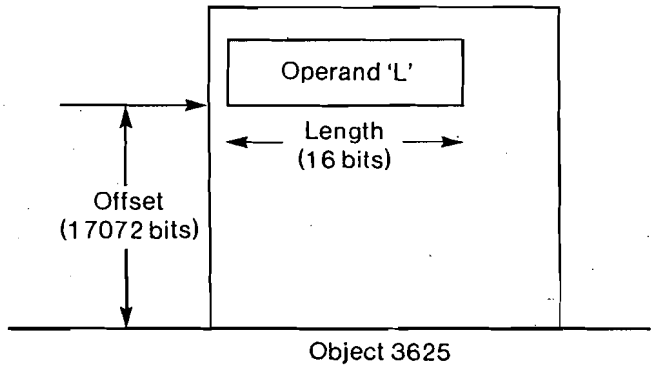
tc38

Figure 3-3. The Short Name Table Entry for Name 123

### 3.1.1.3 Name Table Entry Interpretation

Namespace Addressing now interprets variable L's NTE to produce the variable's logical descriptor. The system examines the NTE Flags field to determine what Namespace Addressing "addressing mode" is to be used. Our example NTE's flags field indicate that it is a short NTE and that the variable is right-justified. The flags field also indicates that the NTE's base value is in the ABR specified in the base field.

The system uses the NTE's base and displacement information to calculate the operand's logical address (UID and offset). The base for our example is the logical address in the Frame Pointer ABR. This means that the operand's value is local to each activation of the procedure and is in the procedure's user stack. The NTE's displacement value indicates that the operand is located 16048 bits from the Frame Pointer ABR's current location. We'll assume that




TC-00040-00

| Flags and Alignment | Type | UID | Offset | Length |
|---|---|---|---|---|
| Right-justified | Integer | 3625 | 17072 bits | 16 bits |

TC-00039-00

3-14
3-17

the logical address in the Frame Pointer is [3625,1024]. This means that the Frame Pointer is located at an offset of 1024 bits into object number 3625. Therefore, variable L's starting location is at (1024 + 16048) = 17072 bits into object 3625.

A logical descriptor consists of an FHP logical address (UID and offset) and the operand's type and length. A system can use this descriptor to address the operand anywhere in the system's logical address space. The logical descriptor for our example operand is shown in Figure 3-4 below. This logical descriptor indicates that the operand is located 17072 bits into the object named by UID 3625. Note that information about the operand that is not used in the UID and offset calculation (length, type, and alignment) is passed through to the logical descriptor. Figure 3-5 shows the location of our example operand in the object.

tc39

Figure 3-4. Logical Descriptor for the Variable 'L'

tc40

Figure 3-5. Location of the 'L' Operand in FHP's Logical Address Space

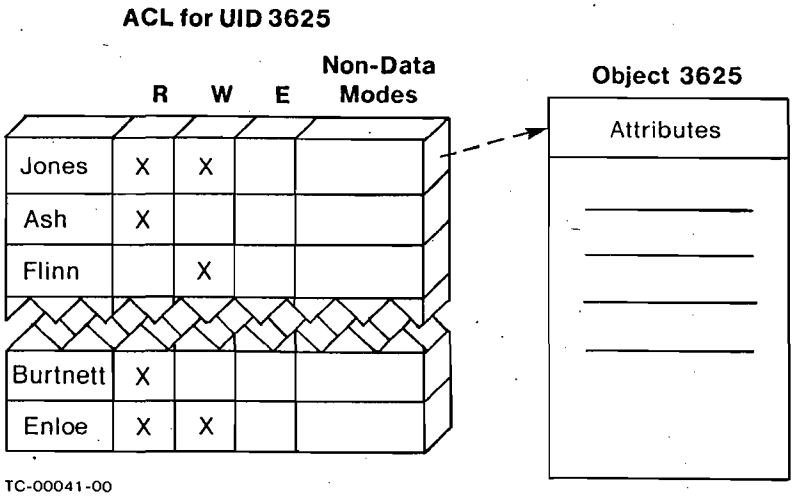
### 3.1.1.4 Validate Subject's Rights to Access the Operand

Protection checks must be performed before a system can actually reference an operand. FHP's protection mechanism examines the subject's access rights using the referenced object's Access Control List (ACL). If we assume that the Jones process is making the reference, then the Jones subject must have read privileges to UID 3625. Information about the object's attributes and Access Control List (ACL) is found in a Logical Allocation Unit Directory (LAUD).

A system error is signalled if Jones does not have read access to the object that contains the value of operand 'L'. In this example, the object is a "user stack" Extended-type Object and the 'L' operand is located in one of its stack frames.

The protection system also examines the operand's length to make sure that the reference does not exceed the bounds of the object. The object's length is one of the attributes found in the

*Jones has read and write access to Object 3625*




TC-00041-00

3-6

Logical Allocation Unit Directory. You are not allowed to reference an operand that extends past the upper bound established by an object's length.

If the protection check fails, no further references are allowed, and the operating system signals a protection fault to the calling process.

tc41

Figure 3-6. Object 3625's Access Control List

The development of a logical descriptor is the last architecturally defined step in FHP addressing. Models of the FHP architecture can use different techniques to translate a logical descriptor's address information into a physical address. We use the logical descriptor shown in Figure 3-4 in a physical addressing example later in this chapter.

| Operation Code | Name for Variable 'B' | Name for Variable 'C' | Name for Variable 'A' |
|---|---|---|---|
| FADD | 229 | 235 | 238 |

| Name for Index 'L' |
|---|
| 237 |

TC-00044-00

### 3.1.2 Referencing an Element of an Array

Now let's look at a more complex, but common, high-level addressing requirement: referencing an element of an vector. Consider the FORTRAN statement:

```
A = B(J,K) + C(L)
```

This statement adds the 'J,K' element of the two-dimensional array 'B' to the 'L' element of the vector 'C'. A simplifieo FORTRAN S-language expansion of this statement is:

```
FADD B (base J,K),C (base L),A
```

Our example demonstrates how Namespace Addressing references the 'L' element of the one dimensional array 'C' (an element of a vector). Notice that a vector reference requires that Namespace Addressing interpret two names: one names the vector; the other names the vector's index.

We assume that our example vector is in the procedure's static data area, which means that it is located using the Static Data Pointer. Recall that the Static Data Area stores data that is used for multiple activations of a procedure.

#### 3.1.2.1 Remove Operand Name From Instruction Stream

The system begins the vector element reference by removing the vector C's name from the procedure object's instruction stream. Let's assume that the numerical "name" the compiler assigned to the variable 'C' is 235.

The compiler also creates an NTE to represent the value of C's index 'L'. We'll assume that this short NTE is indexed by the name 237.

235 is C
237 is L

tc44

Figure 3-7. Names in S-instruction

#### 3.1.2.2 Index Into Name Table to Produce C's NTE

The name (235) of the variable 'C' is used as an index into the procedure's Name Table. The NTE for this example is shown in

| Flags and Alignment | Type | Base | Length | Displacement |
|---|---|---|---|---|
| • Long NTE<br>• IES not a name<br>• Vector | Real | Static Data Pointer | 32 bits | 2048 bits |

| Displacement | Index Name | Reserved | Inter-Element Spacing |
|---|---|---|---|
| N/A | 237 | N/A | 32 bits |

TC-00045-00

Figure 3-8 below. Note that the content of the NTE's extended displacement and reserved fields are not used in this example.

tc45

Figure 3-8. Long Name Table Entry for Name 235

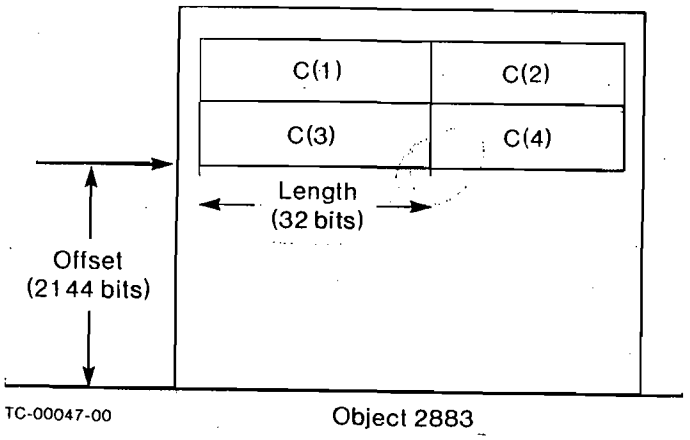
Namespace Addressing uses the NTE's flags field to determine that this is a long NTE, that the NTE represents an element of a vector, and that the Inter-element Spacing (IES) field contains a literal value. The content of the type field is not architecturally defined; the field can be used by S-language interpreters. The type field in our example indicates that the NTE represents a real variable (a floating point number). The NTE's base field indicates that the operand is located in the procedure's static data area. The length field specifies that each 'C' vector element is 32 bits long. The displacement field shows that the first element of the vector is found at a positive displacement of 2048 bits from the logical address in the Static Data Pointer ABR. The Inter-element Spacing field indicates that successive elements of the vector are 32 bits apart. Notice that the index field contains a name (237) that must be interpreted to find which element is being referenced.

### 3.1.2.3 NTE Interpretation

The NTEs for the variable 'C' and the index 'L' are now interpreted to produce a logical address. The first step is to interpret the index NTE to produce the vector element's index value. Interpreting the index NTE proceeds in exactly the same way as the interpretation of our example scalar operand. Let's say that evaluating the index name returns a value of 3. This means that we want to reference C (base 3), the third element of the 'C' vector.

Namespace Addressing now interprets NTE 235 to determine the vector's starting location. This location is 2048 bits from the value in the Static Data Pointer ABR. We'll assume that the Static Data Pointer's value is (UID = 2883 Offset = 0). This means that the vector's first element is located at offset zero in object 2883. The system then multiplies the value of the index (3) by the IES value (32), and adds the result to the vector's starting location. This places our vector element at 2144 bits (2048+(3*32)) into the object.

This two-step NTE interpretation has produced a logical descriptor which locates C (base 3) in object 2883. Figure 3-9


C(1)
C(2)
C(3)
C(4)
Length
(32 bits)
Offset
(2144 bits)
TC-00047-00
Object 2883

OK

3-10

| Flags and Alignment | Type | UID | Offset | Length |
|---|---|---|---|---|
| N/A | Real | 2883 | 2048 + (3*32) bits | 32 bits |

TC-00046-00

shows this vector element's logical descriptor and Figure 3-10 shows the vector's location in the object.

tc46

Figure 3-9. Example Logical Descriptor for C(3)

tc47

Figure 3-10. Location of the Element C(3) in an Object

Protection checking for this example is the same as in the previous example and is not repeated here.

This completes our summary of FHP's architecturally defined addressing and protection features. The following sections describe some non-architectural (model-dependent) techniques that most FHP systems will use to accelerate logical addressing and protection checking. We also describe how typical FHP systems manage and address main memory.

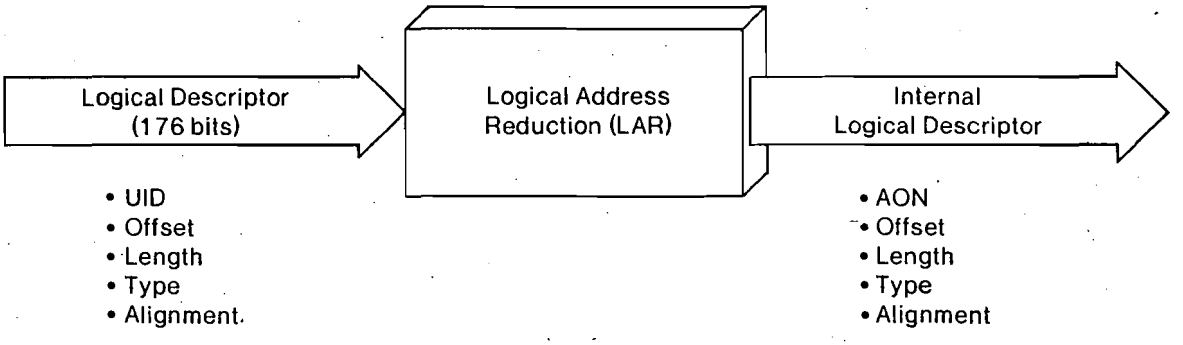
## 3.2 Acceleration Using System Tables

The FHP architecture was designed so that a system's addressing and protection checking operations could be easily accelerated with common hardware and software techniques. This design responds to Data General's goal of developing an architecture that adapts to technological advances. In this section, we demonstrate how FHP systems can use system tables to quickly access frequently used address and protection checking data structures.

Most FHP systems will use a technique called Logical Address Reduction (LAR) to accelerate the architecture's basic object addressing functions. LAR compresses 80-bit UIDs to smaller, more easily managed Active Object Numbers (AONs). Logical Address Reduction uses two system tables as data bases: the Active Object Table and Active Object Hash Table.

FHP systems accelerate protection checking with the same type of technique used to compress UIDs. Protection checking acceleration uses Active Subject Numbers (ASNs) in conjunction with AONs.

The next two sections describe how FHP systems can use per-system tables to accelerate addressing and protection checking.




TC-00029-00

3-11

### 3.2.1 Accelerated Logical Addressing

FHP systems normally use a small subset of FHP's 2**80 available objects. This means that systems can accelerate object references by reducing the number of objects that a system can directly address at one time. These directly addressable objects are called active objects. A system can obtain information about an active object's attributes without referring to a Logical Allocation Unit Directory (LAUD). An active object is identified by its Active Object Number (AON) which temporarily replaces the object's UID. An AON is smaller and easier to manage than the UID that it temporarily replaces. Note that the Logical Address Reduction operation only affects an object's name; no other information within or about the object is changed. For example, the FHP SPRINT uses 14-bit AONs. This means that an SPRINT can directly address up to 2**14 active objects.

AONs are local to a system and are not unique--they are reassigned when the system references an inactive object. Reassignment will generally be made with a least recently used algorithm.

Changing a logical descriptor's UID to an AON creates an internal logical descriptor (see Figure 3-11 below). An internal logical descriptor contains the same information as a full logical descriptor except that the smaller, reusable AON is substituted for the 80-bit UID. Notice that there is no compression of the offset component of logical address; all FHP systems must be able to address bit fields of arbitrary length and location within any object.

tc29

Figure 3-11. Logical Address Reduction (LAR)

Active Object Hash Table and Active Object Table

Logical Address Reduction uses the Active Object Hash Table (AOHT) and Active Object Table (AOT) as data bases (see Figure 3-12 below). An object is active when it has an entry in the Active Object Table. An AOT entry matches a UID to its temporary Active Object Number. An object's AOT entry also contains some frequently used attribute information from the Logical Allocation Unit which contains the object.

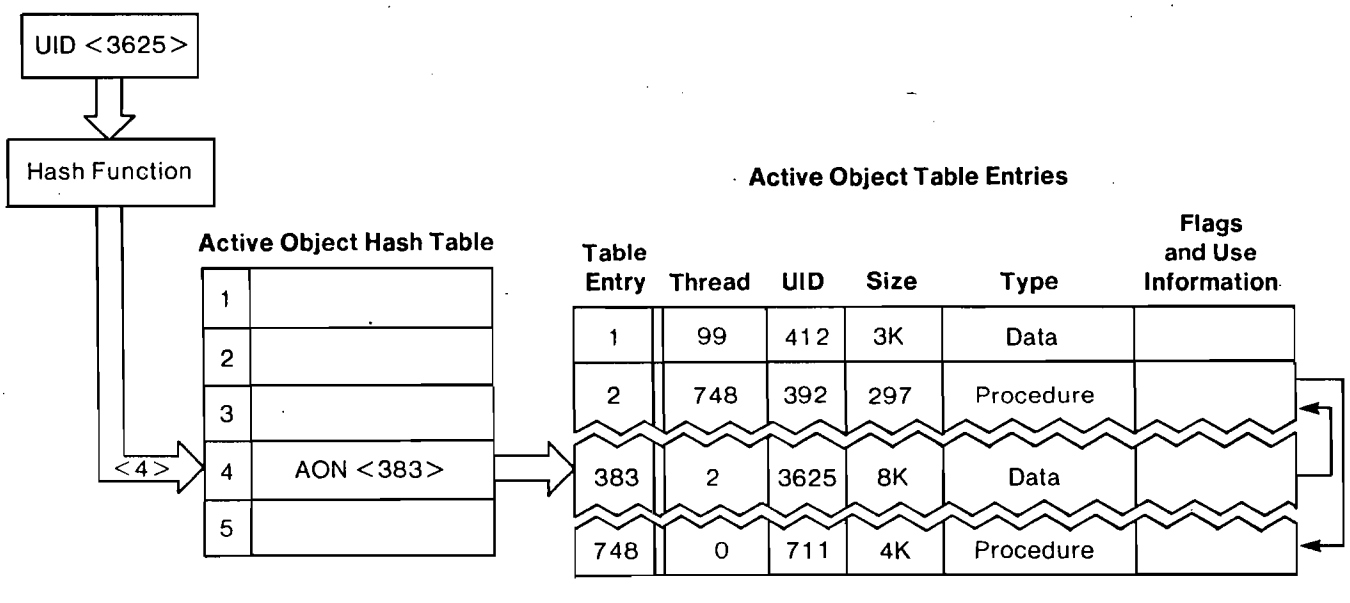
Searches through the Active Object Table are accelerated with a hash coding technique and the Active Object Hash Table. The AOHT

| Flags and Alignment | Type | AON | Offset | Length |
|---|---|---|---|---|
| Right-justified | Integer | 383 | 17072 bits | 16 bits |

TC-00100-00

3-13

12,100




TC-00202-00

3-12

accepts a UID's hash code value and produces the Active Object Number that matches the hash code.

To demonstrate Logical Address Reduction using the AOT and AOHT, assume that an FHP program is referencing the same scalar operand in object 3625 that we used in our earlier addressing example. Remember that the logical descriptor for this operand reference is 176 bits long, with 80 bits used to represent the object's UID. Logical Address Reduction sends UID 3625 through a hashing operation that compresses the UID into a smaller hash code. The UID's hash code is then used as an index into the Active Object Hash Table. In this example, the AOHT entry for the hash code of 4 contains the Active Object Number 383.

The system now indexes the Active Object Table entry for AON 383 and compares the referenced UID to the UID value in the AOT entry. This step is necessary because more than one UID can be hashed to the same hash code number (called a collision). AOT entries have pointers (called threads) to other entries that have the same hash code. Our example indicates that UID 392 has the same hash code as UID 3625. The special null thread (all zeros) in entry number 748 means that this is the last entry for this particular thread.

The UID-to-AON conversion fails if a UID has no Active Object Table entry. This means that the referenced object is inactive. In this case the system automatically deactivates a least recently used object so that its AON may be reassigned. The deactivated object's UID and attributes are removed from the AOT and replaced with the requested object's UID and attributes.

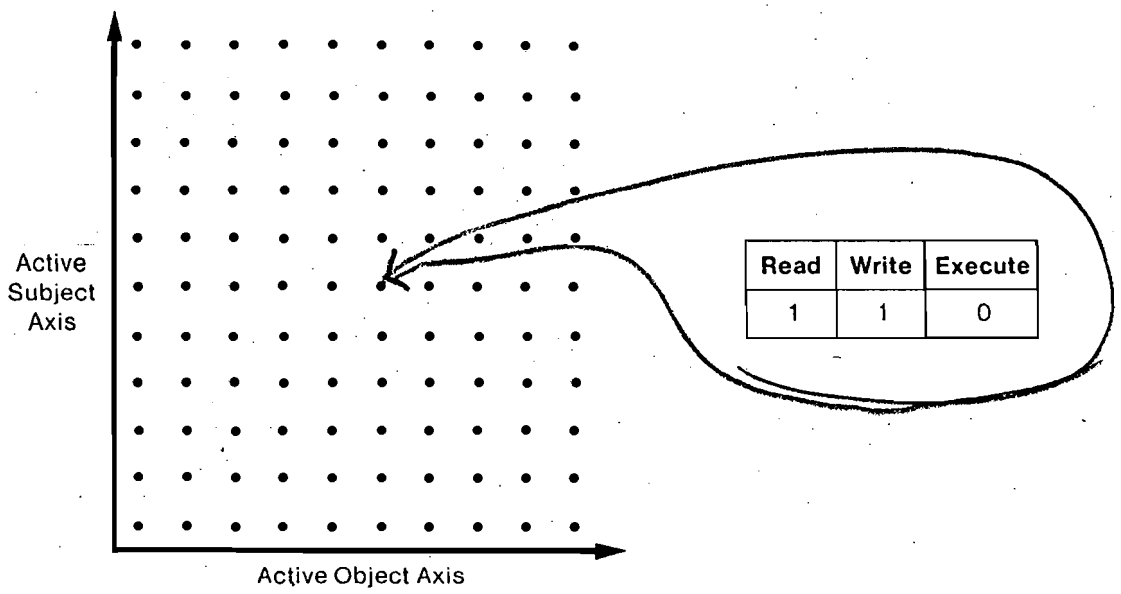
tc202

Figure 3-12. A Logical Address Reduction Example

Figure 3-13 below illustrates the model-dependent internal logical descriptor which would be produced for our example. Notice that no reduction is performed on the operand's starting location (the offset) in the object.

tc100

Figure 3-13. A Typical Internal Logical Descriptor




TC-00015-00

3-14

TC15

### 3.2.2 Accelerated Protection Checking

The ability to quickly determine a subject's access privileges is important to FHP system performance because access rights are checked each time an object is referenced.

An FHP subject is identified by an 80-bit Unique Identifier (UID). Most FHP systems will temporarily assign a smaller Active Subject Number (ASN) to frequently used subjects. Like Active Object Numbers, ASNs are smaller and easier to manage than a subject's full, 80-bit Unique Identifier. Information about an active subject is contained in an Active Subject Table (AST) entry. An AST entry matches an active subject's Unique Identifier with its current Active Subject Number.

An active subject's rights to access an active object are contained in a non-architectural system table called the Active Primitive Access Matrix (APAM). The APAM has active subjects as one axis and active objects as the other axis (see Figure 3-14). The APAM is indexed by active subject and active object to obtain a subject's access rights to a specific object. This access information is taken from the object's Access Control List (ACL). A subject's access rights to an object are checked using the matrix for every object reference. Using Active Subject Numbers and the APAM avoids searching for an object's ACL in a Logical Allocation Unit Directory.

tc15

Figure 3-14. Active Primitive Access Matrix (APAM)

Some models of the FHP architecture further accelerate protection checking with a protection cache. A protection cache contains currently active subject's access rights to a group of objects. The protection cache is discussed later in this chapter.

## 3.3 Physical Addressing

We have seen that FHP's two-dimensional virtual memory contains 2**80 objects. A virtual memory appears to offer the programmer an almost unlimited logical address space. High-level language programmers are not constrained by the size of an FHP system's main memory. An FHP system's main memory is smaller than its logical address space, but the programmer "sees" only the enormous virtual address space.

Present technology and high cost preclude implementing 2**109 bytes of main memory on one system. However, models of the FHP architecture can use non-architectural techniques to manage main (physical) memory addressing.

The architecture requires only that processes generate logical descriptors (UID, 32-bit offset, length, type, and alignment). All FHP systems send these logical descriptors to their memory management systems. The memory management system converts the logical address part of the descriptor (UID and offset) into a physical (main memory) address. The next two sections describe how typical FHP systems manage their main memory and convert logical addresses to physical addresses.

### 3.3.1 Memory Management

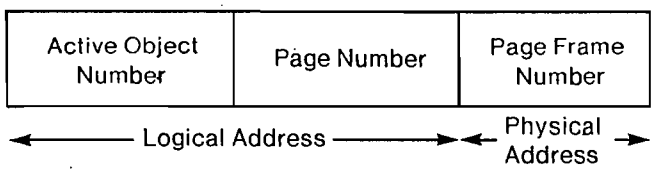
The non-architectural memory management technique used in most FHP systems is called paging. Present technology does not let us deal directly with structures as large as a full-size FHP object (512M bytes). Therefore, FHP systems automatically divide objects into pages, and main memory into page frames. The FHP SPRINT systems' pages and page frames are 2K bytes long (2048 8-bit bytes). For example, an object that is 96K bytes long is divided into 48 pages. A system's virtual-memory manager coordinates the multiplexing, or allocation, of the machine's smaller main memory as a process needs it.

A main memory address in a paged system consists of a page-frame number and an offset into the page frame. For example, a system with 8 megabytes of main memory uses a 26-bit physical address: 12 bits to address its 4096 page frames and 14 bits to address any bit in a page frame.

Because of the potentially large number of pages in each object, FHP systems use demand paging to move requested pages from disk storage pages to main memory page frames. The system automatically moves pages into main memory page frames as they are referenced.

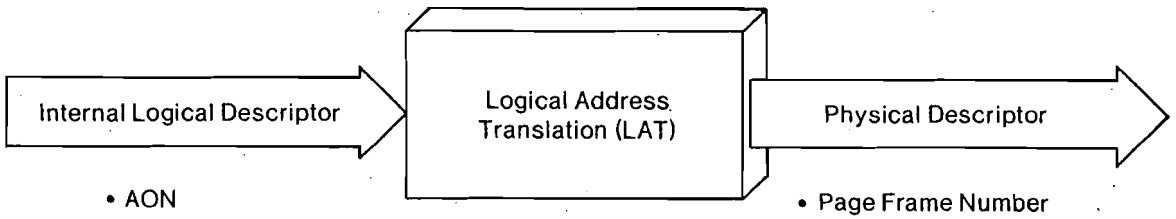
FHP systems can use working sets to control the number of pages a process can have in main memory. A working set establishes a limit on the number of page frames each process can have. Each process is assigned a maximum number of page frames to use at any one time. The size of a process' working set can be set by the system's manager. A process that runs often might be assigned a larger working set than a process that runs infrequently. Working sets let system managers allocate a system's main memory resource equitably, process by process.

OK




TC-00119-00

3-16




- AON
- Offset
- Length
- Type
- Alignment

- Page Frame Number
- Offset
- Length
- Alignment

TC-00030-00

3-15

TC30

## 3.3.2 Logical Address Translation

We have seen how FHP systems can use paging to manage their main memory resource. What we haven't discussed is how FHP systems convert internal logical addresses to physical addresses. This translation is performed by a system's Logical Address Translation function (see Figure 3-15 below). Logical Address Translation (LAT) only proceeds if the subject making the reference has the proper access privileges to an object.

tc30

Figure 3-15. Logical Address Translation (LAT)

Logical Address Translation uses the Memory Hash Table as its data base. Each Memory Hash Table entry contains information about which page is in a page frame. A typical Memory Hash Table entry is shown below in Figure 3-16. Each MHT entry has an AON and page number field that represents an internal logical address, and a page-frame number field that represents the corresponding physical address.

tc119

Figure 3-16. A Typical Memory Hash Table Entry

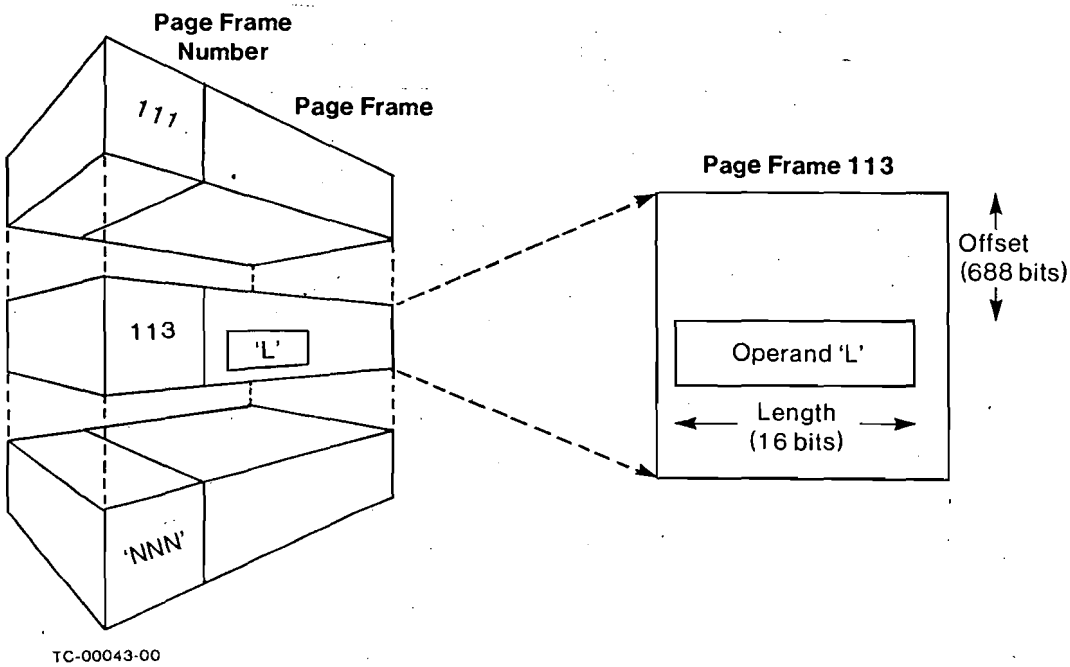
### Logical Address Translation Example

We use the scalar operand example presented earlier in this chapter to give you an example of Logical Address Translation. Recall that Namespace Addressing and Logical Address Reduction produced the following internal logical descriptor for the FORTRAN variable 'L'.

tc39

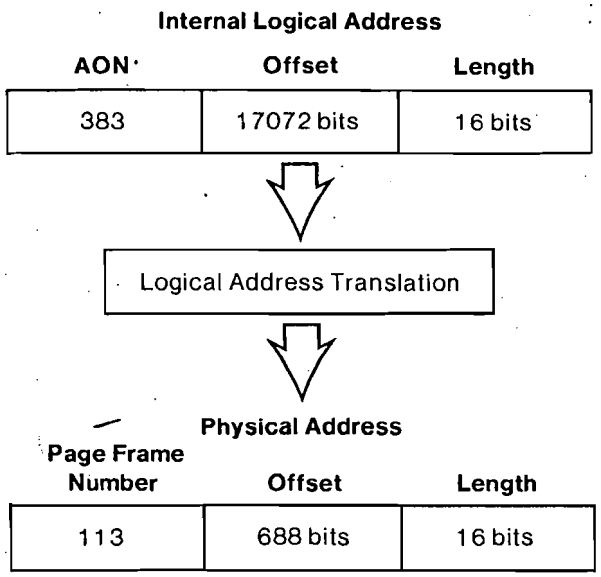
Figure 3-17. Logical Descriptor for the Variable 'L'

Logical Address Translation converts this logical descriptor's address into a physical address as shown in Figure 3-18 below.




TC-00043-00

3-19


Internal Logical Address
AON
Offset
Length
383
17072 bits
16 bits
Logical Address Translation
Physical Address
Page Frame Number
Offset
Length
113
688 bits
16 bits

TC-00120-00

tc120

Figure 3-18. A Logical Address Translation Example

This physical address contains the information required to access the operand in the system's main memory. The physical descriptor has a main memory page-frame number, the operand's starting offset in the page frame, and the operand's length. Notice that the operand's offset reflects the offset in the page frame itself. The operand's starting location is 17072 bits into the object. Each page contains 16384 bits of data. Therefore, the bit-on-page offset is (17072-16384) = 688 bits. Figure 3-19 shows the location of the 'L' variable in a typical system's main memory.

tc43

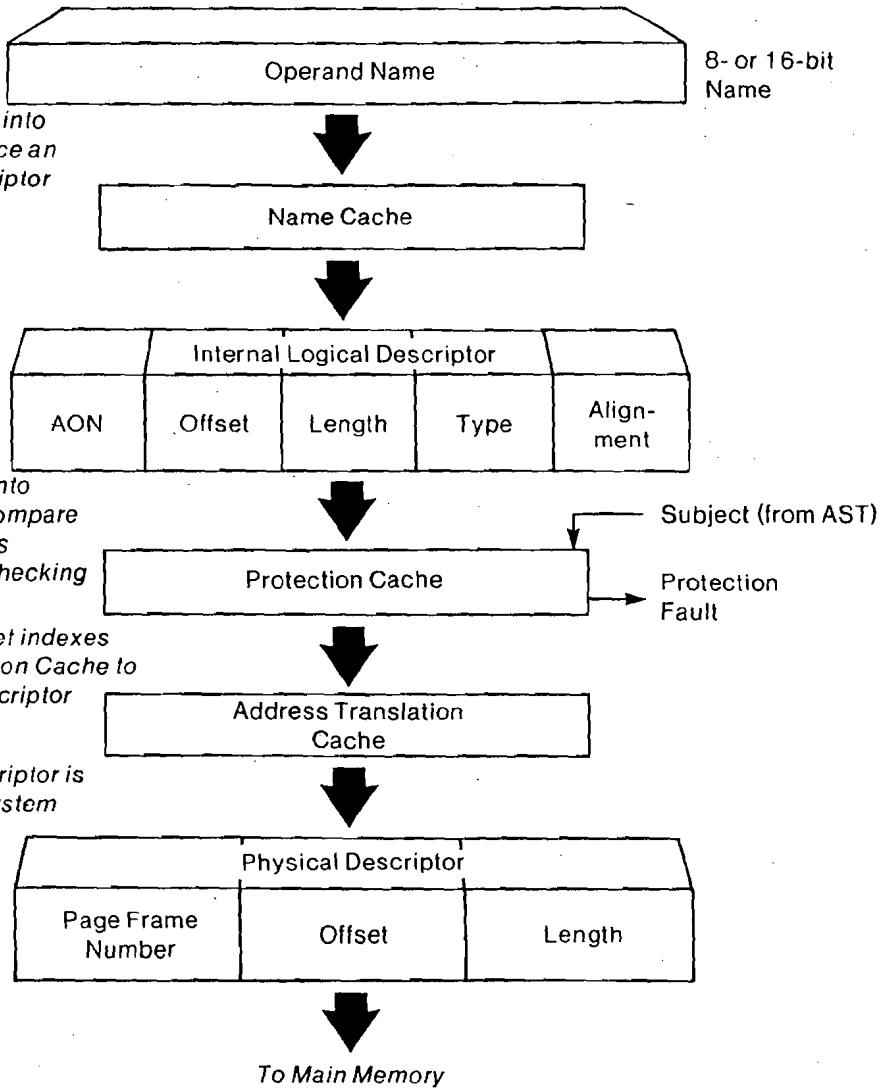
Figure 3-19. The 'L' Operand's Main Memory Location

The Logical Address Translation operation can be accelerated by using an Address Translation Cache. This cache contains information used to convert an internal logical descriptor's AUN, offset, and length to a physical descriptor (page-frame number, page offset and length). We describe the Address Translation Cache in the following section.

Figure 3-20 below is a complete addressing flow chart for a typical FHP system. We have indicated where the logical descriptor creates an interface between architectural and non-architectural functions and data structures. The following section describes how FHP systems can accelerate this basic addressing flow by using cache accelerators.

OK

*Step 1: Remove operand name from S-language instruction stream*

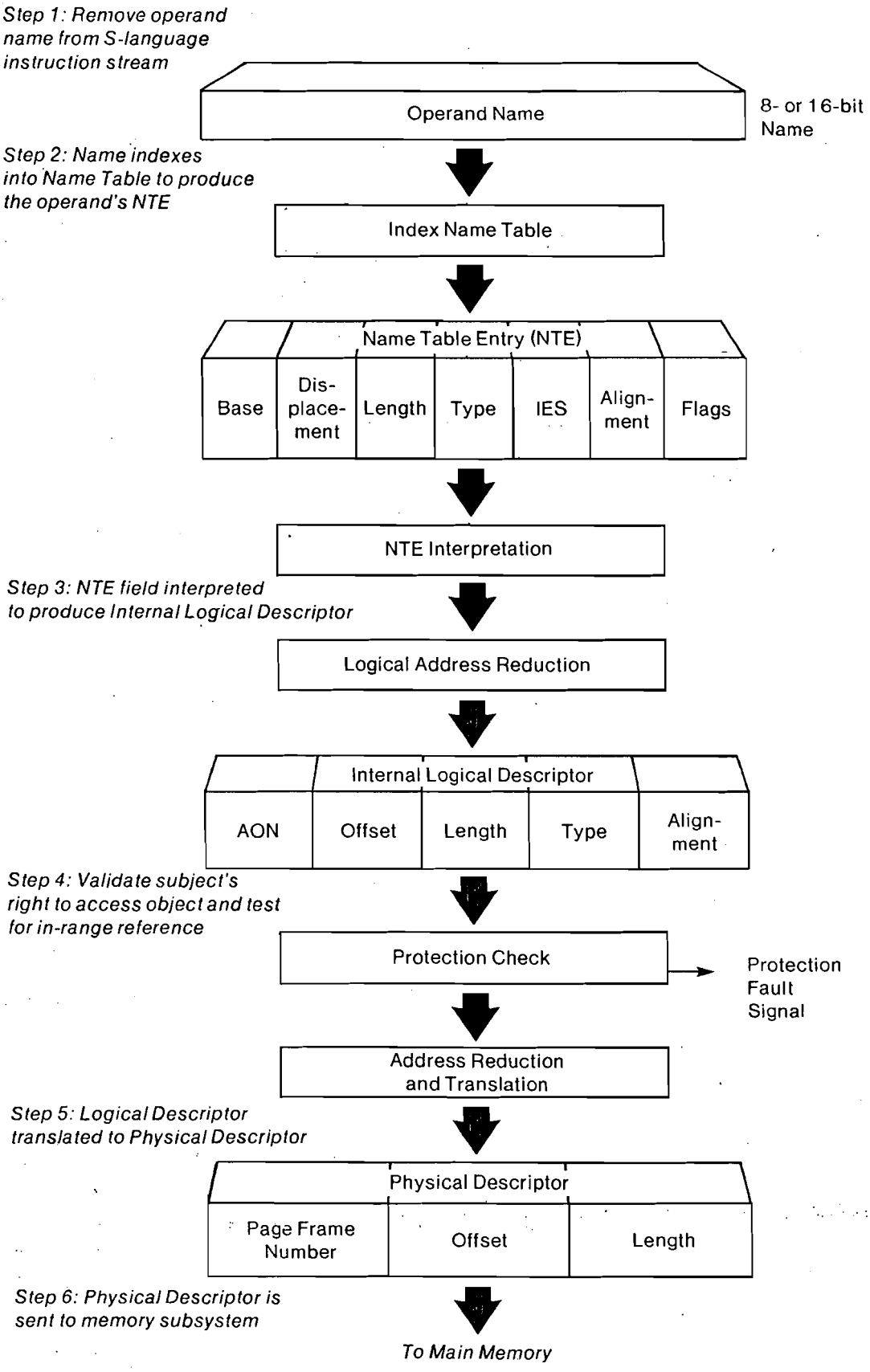



*Step 2: Name indexes into Name Cache to produce an Internal Logical Descriptor*

*Step 3: AON indexes into Protection Cache to compare subject's access rights and perform bounds checking*

*Step 4: AON and Offset indexes into Address Translation Cache to produce Physical Descriptor*

*Step 5: Physical Descriptor is sent to memory subsystem*


Step 1: Remove operand name from S-language instruction stream
Operand Name
8- or 16-bit Name
Step 2: Name indexes into Name Table to produce the operand's NTE
Index Name Table
Name Table Entry (NTE)
Base
Dis-place-ment
Length
Type
IES
Align-ment
Flags
NTE Interpretation
Step 3: NTE field interpreted to produce Internal Logical Descriptor
Logical Address Reduction
Internal Logical Descriptor
AON
Offset
Length
Type
Align-ment
Step 4: Validate subject's right to access object and test for in-range reference
Protection Check
Protection Fault Signal
Address Reduction and Translation
Step 5: Logical Descriptor translated to Physical Descriptor
Physical Descriptor
Page Frame Number
Offset
Length
Step 6: Physical Descriptor is sent to memory subsystem
To Main Memory

TC-00032-00

tc32

Figure 3-20. A Typical FHP System's Addressing Flow

## 3.4 Acceleration Using Caches

The previous two sections described how FHP systems can use tables to accelerate addressing and protection checking. This section describes how FHP systems can use caches to further accelerate these functions.

Various studies show that systems frequently reference a localized group of data while executing a program. Caches are small Random Access Memory (RAM) devices which store this frequently-referenced data. System designers can use caches to increase a system's performance inexpensively if a high percentage of a program's data can be taken from a cache. A cache hit occurs if data is in the cache when requested by the system. A miss occurs if the requested data is not in the cache.

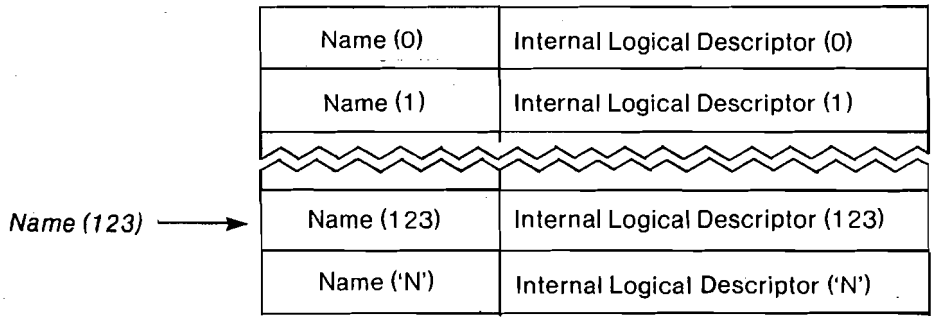
The management of caches and other accelerators is invisible to high-level language programmers, and can be changed from model to model. The programmer is aware of accelerators only as a result of increased system performance.

Figure 3-21 below shows the addressing steps taken in an FHP system using both LAR and caches as accelerators. Caches and other accelerators do not replace FHP's basic addressing functions. These basic functions are used when the required information is not in a cache (a cache miss). This new data is loaded into the cache and is available the next time it is requested.

We discuss three caches in this example: a Name Cache that converts names to internal logical descriptors; a Protection Cache that produces a subject's rights to a specific object, and an Address Translation Cache (ATC) that converts internal logical descriptors into Physical Descriptors. The Name Cache accelerates information from a procedure's Name Table. The Protection Cache accelerates information from the Active Primitive Access Matrix (APAM). The Address Translation Cache accelerates information from the system's Memory Hash Table (MHT).

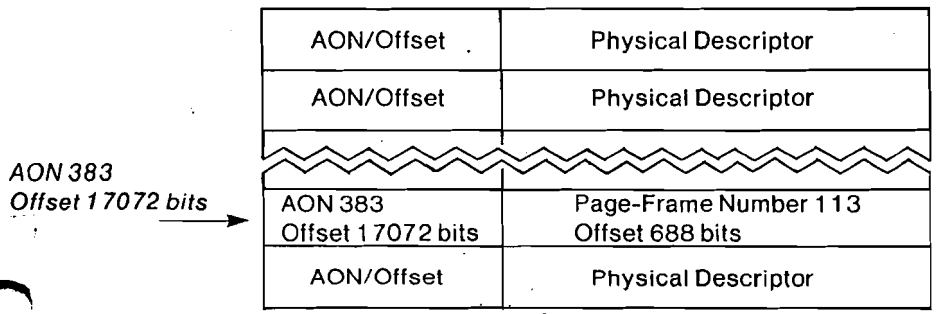
tc101

Figure 3-21. Addressing With Cache Accelerators



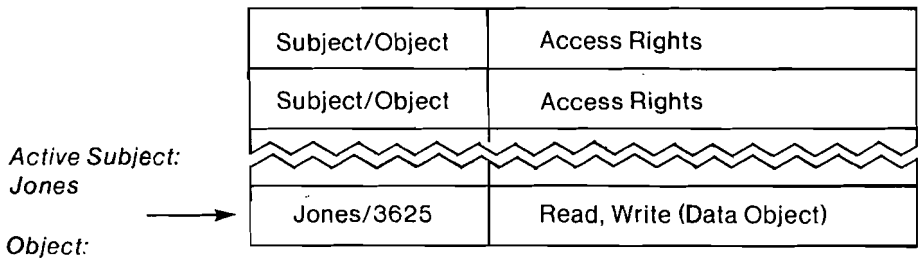

TC-00102-00

3-22




TC-00104-00

3-24




TC-00103-00

3-23

Our example assumes that the required information is available in a cache. If the information is not in the cache, the system reverts to the default path shown before in Figure 3-20.

Notice that Step 1 in Figure 3-21 above is the same as in the unaccelerated example in Figure 3-20. Also, Step 5 in this example is the same as Step 6 in the unaccelerated example. For example, a system using cache accelerators begins an operand reference by removing the operand name for the variable 'L' from the procedure's S-language instruction stream, and ends by sending a physical descriptor to the memory subsystem. However, the number of intermediate steps is reduced from four to three, and each of these three steps is accelerated by a cache.

### 3.4.1 Name Cache

Step 2 uses the Name Cache to bypass the Name Table and the interpretation of the name's Name Table Entry (NTE). The system name for the variable 'L' is used as an index into the Name Cache rather than into the Name Table. A Name Cache entry produces an internal logical descriptor. If the name (123) is in the Name Cache, then its Name Cache entry is the same internal logical descriptor shown earlier in Figure 3-17.

If a Name Cache hit occurs, the NTE interpretation and Logical Address Reduction steps are avoided.

tc102

Figure 3-22. A Typical Name Cache

If the requested name is not in the Name Cache, the system reverts to the Name Table, produces the NTE, and interprets it to produce the logical descriptor. Then, Logical Address Reduction replaces the logical descriptor's UID with an Active Object Number, to produce the internal logical descriptor.

### 3.4.2 Protection Cache

Step 3 bypasses an Active Primitive Access Matrix (APAM) reference by using the Active Object Number component of the internal logical descriptor as an index into the protection cache. The protection cache is indexed by Active Object Number and and Active Subject Number to produce an active subject's rights to reference frequently used active objects. If the calling subject's rights to the object are in the cache, the rights are checked, and

the operand's length information is compared to the length of the object. If either protection check fails, a protection fault is signalled, and operand addressing is terminated at that point.

tc103

Figure 3-23. A Typical Protection Cache

If the protection cache misses, the basic protection checking mechanism is used and the APAM is examined.

### 3.4.3 Address Translation Cache

If the protection checks are positive, the Address Translation Cache (ATC) can be indexed by AON and offset to produce a physical descriptor directly. The Address Translation Cache avoids a Logical Address Translation search of the Memory Hash Table. Address Translation Cache references can actually be started while the protection checks are being made.

A hit in the ATC generates the operand's physical descriptor, which is sent to main memory. Logical Address Translation (LAT) is used to form the physical descriptor if the ATC misses.

tc104

Figure 3-24. An Address Translation Cache

## 3.5 Conclusions

We have described FHP's major features, and given examples of how these features are integrated into the architecture.

As an FHP system user, you have a system that:

* offers better price/performance than conventional systems,
* is easy to use, and
* supports your long-term data processing requirements.

Table 3-1 lists the FHP design goals introduced in the first chapter and summarizes the features that meet each goal. The section following the table shows you how these features contribute to FHP's price/performance, ease of use, and longevity.

Table 3-1. Summary of FHP's Features

| | | |
|---|---|---|
| PRICE/PERFORMANCE<br>EASE OF USE<br>LONGEVITY | High-level Language Efficiency | *S-languages<br>*Dynamically switched S-language interpreters |
| | Addressing Flexibility | *Universal, object-oriented address space<br>*Namespace Addressing |
| | Security | *Access Control List for each object<br>*Domains to administer a subject's authority |
| | Technological Flexibility | *Memory-intensive design<br>*Acceleration potential |

## Price/Performance

FHP is a very flexible architecture and can support systems that span a broad range of performance. FHP's universal, object-oriented address space is one of its most powerful features. FHP's universal address space contains 2**80 uniquely identified and protected objects. Each object can contain up to (2**32)-1 bits of information. All data and procedures are contained in objects. FHP systems have two types of object: primitive and extended-type.

Extended-type Objects let system programmers create abstract data structures and define the operations that can be performed on them. In Chapter 2 we described the use of Extended-type Objects to create stack objects. FHP systems also define other Extended-type Objects, such as processes, domains, and subjects. FHP's Extended-type Objects are also useful in data base and communication applications.

An FHP system's S-language interpreters are usually implemented inhost-machine microcode. FHP systems can use different host machines and host-machine microcode without affecting the higher-level S-language interfaces. S-language interpreters can be dynamically switched into the host machine's Writable Control Store as processes execute different high-level language programs. A high-level language running on an FHP system "sees" a processor that is designed to efficiently support that language's requirements.

The FHP architecture is designed for modular implementation. Most FHP systems will contain a small number of well-defined hardware and firmware subsystems. One subsystem will support the requirements of the FHP host machine. Other independent subsystems can perform concurrent Input/Output and diagnostic functions.

An FHP system's pointers and system tables could result in powerful, but slow systems. However, FHP's memory-intensive design means that models of the architecture can use different acceleration techniques to tailor a system's performance. High-performance FHP systems will use faster and larger caches to accelerate frequently used data. Other FHP systems can use smaller or fewer accelerators to meet specific performance goals.

## Ease of Use

FHP programmers need not be concerned with the size of their application programs or data structures. FHP objects are automatically divided into pages that are moved into a system's main memory on demand. This means that more processes can share a system's valuable main memory resource.

FHP's S-languages let programmers prepare system software using application-efficient languages. S-language instructions are compact, powerful, yet easy to use. Each S-language can have 256 instructions, tailored to the requirements of different high-level languages. S-languages are easy to use because programmers need not be concerned with managing a complex set of host-machine registers.

You have seen that FHP's security system is powerful and flexible. It is also easy to use. All FHP objects are protected by Access Control Lists (ACLs). An FHP process performs work for a specific user (a subject), and an object's ACL contains each subject's rights to perform operations on that object. Programmers can prepare specialized protection sub-systems for their own applications without disturbing other user's applications.

## Longevity

The FHP architecture is designed to be both rich and flexible. The architecture does not restrict implementations to a single hardware or software technology.

FHP's memory-intensive architecture provides individual data structures and tables that are easily accelerated by small, fast-access memories. FHP systems will continue to offer excellent price/performance ratios as the cost of semi-conductor memories declines.

It is important to remember that an FHP system's accelerators are not defined as part of the FHP architecture. A system's accelerators are hidden from high-level language programmers. It is also important to remember that FHP S-language programs use logical addresses made up of Unique Identifiers and offsets. However, the architecture does not restrict the use of different kinds of physical addresses. Therefore, FHP systems can take advantage of future developments in accelerator and physical addressing technology without affecting the high-level language programmer's interface.

The architecture can also keep pace with advances in software technology. FHP's S-languages and dynamically switched S-interpreters combine to support a wide range of current and future high-level languages. FHP developers have created a powerful system programming language for FHP systems. This language, called SPL, is similar to PASCAL. SPL supports structured programs and provides strong data typing.

Please consult the FHP SPRINT Publications Guide (069-600000) if you want more information about the FHP SPRINT documentation set.

--End of Chapter--

# Appendix A
# Glossary

## Acronyms for FHP terms in this Glossary

| | |
|---|---|
| ABR | Architectural Base Register |
| ACL | Access Control List |
| AOHT | Active Object Hash Table |
| AON | Active Object Number |
| AOT | Active Object Table |
| APAM | Active Primitive Access Matrix |
| ASN | Active Subject Number |
| AST | Active Subject Table |
| DOE | Domain of Execution |
| EACL | Extended Access Control List |
| ED | Entry Descriptor |
| ETM | Extended-type Manager |
| FP | Frame Pointer |
| IES | Inter-element Spacing |
| k | Operand syllable size |
| LAR | Logical Address Reduction |
| LAT | Logical Address Translation |
| LAU | Logical Allocation Unit |
| LAUD | Logical Allocation Unit Directory |
| LAUID | Logical Allocation Unit Identifier |
| NTE | Name Table Entry |
| NTP | Name Table Pointer |
| OSN | Object Serial Number |
| PBP | Procedure Base Pointer |
| PC | Program Counter |
| PED | Procedure Environment Descriptor |
| RAM | Random Access Memory |
| SDP | Static Data Pointer |
| UID | Unique Identifier |
| VP | Virtual Processor |
| WCS | Writable Control Store |

Accelerate

To speed up data transfers within a computer system. One kind of acceleration moves data to a more accessible level in a system's storage hierarchy. A memory data cache is an example of traditional acceleration. Frequently accessed data is moved from memory to the faster access cache.

Data transfers can also be accelerated by overlapping data fetches and data manipulation (see Pipelining), or with software-based hash coding techniques.

Access Control List

(ACL)

An attribute of an object that specifies which subjects can perform system-defined operations on the object. These operations include read, write, and execute, as well as non-data operations. Non-data mode operations include the right to delete the object or modify its attributes.

Extended-type Objects have a second access control list, the Extended Access Control List (EACL). An EACL specifies which subjects can perform user-defined operations on an object. Extended operations might include "push" and "pop" for an Extended-type Object of type "stack."

Active object

An object that an FHP system can address without referring to a Logical Allocation Unit Directory. An active object is described in the system's Active Object Table, where the object's Unique Identifier (UID) is associated with a smaller Active Object Number (AON).

Active Object Hash Table

(AOHT)

A table that can be used to accelerate searches in the Active Object Table. The Active Object Hash Table accepts a hash coded (compressed and encoded) version of an Unique Identifier (UID). The table produces the Active Object Number corresponding to the UID's hash code. The Active Object Number is then used as an index into the Active Object Table.

Active Object Number

(AON)

The lowest-level, most transient name of an object. FHP systems can temporarily replace an object's UID with a

smaller, easily manageable Active Object Number. The system can then address the object by its AON. AONs are system-specific and are reused within a system. An Active Object Number indexes the Active Object Table to produce the object's attributes.

Active Object Table
(AOT)
A non-architectural, per-system table in main memory. The AOT accelerates access to selected object attributes. This table is indexed by AON to produce an active object's accelerated attributes.

Active Primitive Access Matrix
(APAM)
A memory resident matrix that is indexed by subject and AON. The APAM accelerates a subject's access rights to a primitive object. These access rights are in a subject's Access Control List entry.

Active Subject Number
(ASN)
A per-system acceleration of a subject's Unique Identifier (UID). Active Subject Numbers are smaller than 80-bit UIDs and are easier to manage. An operating system assigns Active Subject Numbers to recently used subjects.

Active Subject Table
(AST)
A per-system table that contains a list of recently used subjects. An Active Subject Table entry associates a subject's UID with its temporary Active Subject Number.

Address
An object's location in memory. An address, when supplied to a memory, selects a subcomponent of the memory. An FHP address specifies one bit in FHP's universal address space. An FHP address uses an 80-bit UID to locate an object and an 32-bit offset to locate any bit in the object.

Address Translation Cache
(ATC)
Most FHP systems will use an Address Translation Cache to accelerate logical-to-physical address conversions. This non-architectural cache is indexed by logical address (AON

and offset) to produce a physical address (page-frame number and offset on the page).

Architectural Base Register
(ABR)
One of three architecturally-defined pointers to storage locations in FHP's universal address space. FHP's ABRs are the:

* Procedure Base Pointer (PBP),

* Frame Pointer (FP), and

* Static Data Pointer (SDP).

An FHP operand's logical address is found using a base/displacement calculation. The base for this calculation is the logical address in one of the Architectural Base Registers. ABR values are part of a process' macro-state.

Architectural Call
FHP's Architectural Call performs secure procedure-to-procedure and procedure-to-system calls. Architectural Call automatically checks access rights and saves information needed to return from the call. The Architectural Call mechanism is common to all FHP S-languages.

Architectural Call accepts the target procedure's UID and any arguments that will be passed to the target procedure. Architectural Call automatically pushes the current macro-state onto the user's current stack, adds a new stack frame to the target procedure's stack, and transfers control to the target procedure. If necessary, Architectural Call switches a process to a different S-interpreter and changes domains (a cross-domain call).

Architectural Return
Architectural Return restores the environment that was in effect before an Architectural Call. This may entail switching a process' domain of execution and current S-interpreter.

Architecture

A guaranteed user interface to a system. An architecture presents different interfaces to different users. A high-level language programmer's interface is established by the system's compiler and operating system. Data General's FHP architecture consists of a collection of interacting interfaces which provide features such as a 112-bit, object-oriented address space; Namespace Addressing; domain protection; and dynamically switched, S-language interpreters.

Automatic data

A type of data whose storage is unique to each activation of a procedure. FHP systems store automatic data on the current user stack. Automatic data includes temporary variables that are not used by any other procedures.

Base

A logical address which begins each FHP address calculation. Operand addresses are obtained with a base/displacement calculation. The base in these calculations is a value from one of the Architectural Base Registers.

Cross-domain call

A type of Architectural Call that causes a process' domain of execution to change. A cross-domain call occurs when the target procedure is in a different domain from the calling procedure.

Cross-domain stack frame

A collection of information that is pushed onto a process' secure stack during a cross-domain call. This information is protected, to allow a secure return to the calling procedure's domain.

Data object

One of the four types of FHP primitive object. Data objects may contain from 0 to (2**32)-1 bits; they have no architecturally-defined internal structure.

Demand paging

Most FHP systems will use demand paging to reduce the number of page frames a process has in main memory. A page is moved into a main memory page frame only when referenced

by a process. Unreferenced pages remain in backing store, allowing efficient use of the main memory resource.

Descriptor

A data structure that describes and locates an FHP object. Descriptors contain an operand's address and its type, length and alignment. FHP systems use two types of descriptor: logical and physical. Logical descriptors contain an operand's logical address (Unique Identifier (UID) and offset). Physical descriptors contain an operand's physical address in the form of a page frame number and offset on the page frame.

Displacement

Used in Namespace Addressing's base/displacement calculations. The base part of this calculation is a logical address from an Architectural Base Register (ABR). An operand's offset in an object is calculated by adding the operand's Name Table Entry displacement value to an ABR's offset value.

Domain

Part of FHP's security system. A domain represents a set of rights to access information in an FHP system. A process is always associated with a domain. A process' current domain helps determine the process' authority. Domains are defined as Extended-type Objects, and are named by the Extended-type Object's Unique Identifier (UID).

Domain of Execution
(DOE)

An attribute of a procedure object. If set, the Domain of Execution names a domain. A cross-domain Architectural Call is performed if a procedure is called from a domain other than its specified DOE. This call automatically switches a process' protection regime.

Dynamic link

A non-resolvable pointer that represents an object's logical address. A pointer has a fault bit that specifies that the pointer cannot be resolved by Namespace Addressing. A fault is signalled when Namespace Addressing tries to make a reference through a non-resolvable pointer. This fault invokes a special operating or language system dynamic-link fault handler. The fault handler transforms the pointer's symbolic address into a real logical address.

Entry Descriptor
(ED)
Each FHP procedure is identified and located by its Entry Descriptor (ED). A procedure can be called from outside its procedure object if the procedure's ED is located with the object's gate list. Otherwise, the procedure can only be called from within its own object.

Extended Access Control List
(EACL)
A list that specifies a subjects' right to perform operations on an Extended-type Object. See Access Control List.

Extended-type Manager
(ETM)
One of FHP's four primitive object types. Extended-type Managers contain procedures that perform operations on Extended-type Objects. ETMs have the same basic structure as procedure objects. ETMs have special modes of access that control the ability to create and delete Extended-type Objects. An ETM can manage more than one Extended-type Object.

Extended-type Object
(ETO)
An FHP object that can be used to implement secure, abstract data structures. System programmers can place Extended-type Objects into domains so that data in the objects can be manipulated only by the object's Extended-type Managers.

Frame Pointer
(FP)
One of FHP's three Architectural Base Registers. The Frame Pointer locates a procedure's activation record and automatic data.

Gate
An entry point into a procedure object. A gate is either a UID pointer locating another gate or an Entry Descriptor. Procedure objects may only be entered by calling through a gate contained in the procedure's gate list. This list immediately follows a procedure object's header.

Hash coding
: A software-oriented acceleration technique used to speed up searches in tables. FHP systems can use hash coding compression techniques to access data bases such as the Active Object Table (see Active Object Hash Table).

Host machine
: FHP systems are implemented using model-dependent host machines. A host machine is a collection of hardware that is enhanced with software and firmware to create a set of Virtual Processors. The FHP architecture does not restrict the type of hardware or firmware technology used to implement a model's host machine.

Instruction
: The smallest unit of control for an FHP processor. An FHP S-language instruction consists of an 8-bit operation code (see Opcode), followed by some number of operand syllables. The Opcode defines the instruction's syntax and semantics. Operand syllables can be operand names, relative branches, or literals. An operand syllable can be eight or sixteen bits long. The operand syllable size, 'k', is specified in a procedure's Procedure Environment Descriptor (PED), and remains constant throughout that procedure.

Instruction stream
: (I-stream)
A series of S-language instructions that direct the execution of an FHP processor.

Instruction stream syllable
: A unit of information in an S-language instruction stream. There are two basic kinds of instruction stream syllable: opcode syllables and operand syllables. All opcode syllables are 8 bits long. An instruction's operand syllables must be the same length (see k).

Inter-element Spacing
: (IES)
A field in the Name Table Entry extension that indicates the number of bits between successive elements of a vector. Namespace Addressing uses the IES value to calculate the location of an element in an vector.

Inter-object pointer

A 128-bit data structure that contains a logical address (UID and offset), and a 16-bit flags and format field. An Inter-object pointer can address any bit in the FHP address space. Inter-object pointers may be contrasted with Intra-object pointers. Note: Inter-Object pointers are also called UID pointers.

Internal logical descriptor

A model-dependent data structure created by Logical Address Reduction. An internal logical descriptor contains information that locates and describes an S-language operand. The descriptor has a logical address in the form of Active Object Number (AON) and offset as well as the operand's length, type, and alignment. Logical Address Translation converts a logical descriptor to an internal logical descriptor by substituting an Active Object Number for the logical descriptor's UID.

Intra-object pointer

A short-format FHP pointer. An Intra-Object pointer is 48 bits long, and can locate data only within the object in which the pointer resides. This pointer consists of a 32-bit offset into its object, and a 16-bit flags and format field. This field includes information about the pointer's type. Note: Intra-object pointers are also called an Object Relative pointers.

k

'k' represents the number of bits in an S-language instruction stream operand syllable. 'k' can be either 8 or 16 bits. k's value is specified in a procedure's Procedure Environment Descriptor (PED).

Literal syllable

An operand syllable that is not interpreted by Namespace Addressing. A literal syllable is treated as a signed, 'k' bit value.

Logical address

A location in FHP's universal address space. FHP systems use two types of logical address: internal and external. Each type has two fields: an object identifier and a 32-bit offset into the object. The object identifier in a non-architectural internal logical address is an Active Object Number (AON). The object identifier in an external logical

address is a Unique Identifier (UID).

Logical Address Reduction
(LAR)
The non-architectural function that translates an external memory address (UID and offset) into an internal memory address (AON and offset). Logical Address Reduction uses the Active Object Table as a data base. If the object designated by the UID is not active, an inactive object fault is signalled. A fault handler then releases the least-recently used object and assigns the AON to the new object.

Logical Address Translation
(LAT)
A non-architectural function that translates an internal logical address (AON and offset) in to a physical address (page-frame number and offset). Logical Address Translation uses the Memory Hash Table as its data base. LAT can be accelerated with an Address Translation Cache.

Logical Allocation Unit
(LAU)
A permanent storage container for all of a system's objects. Each LAU has a directory which describes each object in the LAU. A Logical Allocation Unit Directory (LAUD) is indexed by UID to produce an object's attributes. A system's Active Object Table accelerates selected data from Logical Allocation Unit Directories.

Logical Allocation Unit Identifier
(LAUID )
A 32-bit identifier, unique in time and space, that names a Logical Allocation Unit (LAU). An LAUID is part of a field in an object's Unique Identifier (UID). Data General assigns part of an LAUID to a system to guarantee that UIDs remain unique from system to system.

Logical descriptor
A data structure that locates and describes an S-language operand. Logical descriptors are produced when a Name Table Entry is interpreted. A logical descriptor contains an operand's location (UID and offset), and the operand's length, type, and alignment. The model-dependent Logical Address Reduction function converts a logical descriptor to an internal logical descriptor by substituting an Active

Object Number for the logical descriptor's UID.

Logical Input/Output
: The FHP function that provides device-independent I/O system support to user and system programs.

Long Name Table Entry
: A 128-bit Name Table Entry, describing a vector operand or an operand that displaces more than sixteen bits. A long NTE consists of two contiguous, 64-bit Name Table elements.

M bytes
: A suffix used to express memory size by powers of 2. An M byte is 2**20 bytes. Therefore, 1M byte is equal to 1,048,576 bytes.

Macro-state
: Information that represents the state of a processor when a procedure is deactivated. Macro-state consists of a procedure's current ABR values, the offset portion of the PC, the offset of the top of the procedure's activation record, and the location of the procedure's Entry Descriptor. The Name Table Pointer, located by the Entry Descriptor, is implicitly part of macro-state. Most FHP systems store macro-state on a protected stack object called the secure stack. Macro-state is held on the secure stack to guarantee proper returns from calls.

Main memory
: An FHP system's physical memory. Most FHP systems implement main memory with fast-access semiconductor storage. Main memory size is system-dependent. A system's main memory is divided into page frames and accessed with physical addresses consisting of a page-frame number and an offset on the page. An FHP system's main memory is bit addressable.

Micro-state
: The internal state of a Virtual Processor and its supporting host machine. Micro-state is stored when a process faults or is pre-empted. The process is restarted by reloading its micro-state. Most FHP systems will store micro-state in the protected secure stack.

Name

An S-language operand syllable that indexes a procedure's Name Table to produce a Name Table Entry. Names can be taken from an instruction stream or certain Name Table Entry fields. S-language instructions require exactly one name to specify a variable reference.

Name evaluation

One of Namespace Addressing's basic functions. Name evaluation automatically transforms an 8-bit or 16-bit operand name into the operand's value, length, and type.

Name resolution

One of Namespace Addressing's basic functions. Name resolution automatically transforms an 8- or 16-bit operand name into the operand's logical descriptor (logical address, length, alignment, and type).

Name Table

A vector of up to 2**16 64-bit elements (Name Table Entries), indexed by names. Name Table Entries may occupy one or two contiguous elements of a Name Table. (See Name Table Entry). A procedure's Name Table is created when the procedure is compiled. Procedures may share Name Tables.

Name Table Entry

(NTE)

A data structure which contains information used to calulate an FHP operand's description. A Name Table Entry can specify other NTEs, whose resolution contributes to the resolution of the parent entry. A Name Table Entry occupies either one or two contiguous, 64-bit Name Table elements.

Name Table extension

The additional, 64-bit element of a Name Table containing the last half of a Long Name Table Entry. The extension includes the high-order 16 displacement bits, and index and Inter-element Spacing fields.

Name Table Pointer

(NTP)

A pointer to the starting location of a Name Table. The current Name Table Pointer is a part of a process' macro-state.

Namespace Addressing

FHP's operand-addressing acceleration technique that is used by all FHP S-languages. Namespace Addressing transforms short (8-bit or 16-bit) operand names into operand values or locations with two basic operations: name resolution and name evaluation. Resolving a name automatically returns the named operand's location and description. Evaluating a name returns the named operand's value, length, and type.

Object

The basic unit of storage in the FHP architecture. Objects store up to (2**32)-1 bits of data. Objects are described by their attributes. An object's attributes include its length, type, Access Control List, and associated information, such as time-stamps. The four primitive object types recognized by the FHP architecture are data, procedure, S-interpreter, and Extended-type Manager.

Object Serial Number
(OSN)

A 48-bit number that is part of an object's Unique Identifier (UID). An OSN identifies an object within a Logical Allocation Unit (LAU). OSNs are unique; they are never reused within an LAU. All objects in an LAU have the same 32-bit Logical Allocation Unit Identifier.

Offset

The address of a bit within an object. Offsets support FHP's bit-granular addressing.

Opcode

The instruction syllable defining the semantics of an operation such as add or subtract. An opcode can also specify the syntax and number of operand syllables in an instruction. FHP S-language opcodes are eight bits long.

Operand syllable

An instruction stream token, 'k' bits long, associated with a particular Opcode. FHP operand syllables follow the Opcode in logically contiguous memory. Operand syllables may be operand names, address branch offsets, or literal values.

Page

An address space subdivision. An FHP system automatically divides objects into pages, to efficiently manage the system's main memory. Current FHP systems use 2K-byte pages (2048 bytes). A 64K-byte object would be divided into 32 pages.

Page frame

An FHP system's main memory is divided into page frames. An FHP system's memory manager automatically moves pages from backing store to main memory page frames (see Demand paging).

Physical address

An address in main memory. An FHP system's physical address contains a page-frame number and an offset into the page frame. Physical addresses are produced by Logical Address Translation.

Pipelining

A non-architectural acceleration technique. Pipelining uses buffers to overlap two operations. Most FHP systems use pipelining to fetch an instruction while the previous instruction is executing.

Pointer

A pointer contains or represents a logical address, and locates a bit in FHP's address space. Pointers have two sizes: 128-bit, inter-object pointers that address any bit in FHP's memory; and 48-bit, intra-object pointers that locate any bit in the object containing the pointer. Both pointer types contain a 16-bit flags and format field. This field identifies the pointer's type and indicates whether the pointer can be resolved by Namespace Addressing.

Primitive object

One of two basic types of FHP object. There are four kinds of primitive object: procedure, data, S-interpreter, and Extended-type Manager.

Principal

Principal is one component of a subject. A principal identifies the name of the user or group of users that a process is working for.

Procedure Base Pointer
(PBP)
One of the three Architectural Base Registers. The Procedure Base Pointer (PBP) locates the beginning of a procedure's S-language code. The PBP is used to address program constants and initial values and calculate absolute branch addresses within a procedure object.

Procedure Environment Descriptor
(PED)
A data structure that describes the environment a procedure needs when it executes. The Procedure Environment Descriptor specifies the required S-interpreter, and locates the procedure's Name Table and Static Data storage area. The PED also specifies the procedure's operand syllable size, 'k'.

Procedure object
One of FHP's four primitive object types. Procedure objects differ from other objects in that they have gates and can be called by other procedures. Procedure objects usually contain I-streams, Entry Descriptors, Name Tables, literal values, and program constants.

Procedure object header
A 128-bit data structure, located at offset zero of a procedure object. The procedure object header describes the object's structure and the number of gates which follow the header in the procedure object.

Process
An operating system data structure. An FHP process is an asynchronous flow of S-language code execution through procedure objects. An operating system's process manager determines what system resources are required to execute a program or group of programs, and makes those resources available on a shared basis.

A process represents the state of the system as a program executes. By saving state information, the operating system can stop a process, then restart it by restoring its state.

Processes have attributes that control their ability to access information and to use system resources. Each FHP process has a Unique Identifier (UID) and competes with other processes for system resources such as Virtual

Processors, memory and I/O devices.

Program
: A series of instructions that control a computer's operation.

Random Access Memory
(RAM)
: A data storage medium that provides read and write access to data. Data transfers to and from a RAM occur at the same speed, regardless of the data's location in the RAM. The FHP SPRINT uses fast-access, semiconductor RAMs in its memory arrays and caches.

Relative branch syllable
: An operand syllable specifying a location relative to an S-language instruction stream's Program Counter (PC). The PC points to the leftmost bit of the opcode being interpreted.

S-interpreter
: One of the four types of FHP primitive objects. S-interpreter objects contain the programs which interpret S-language instructions. S-interpreters are usually implemented in host-machine microcode.

S-language
: An intermediate system language. An FHP S-language is a memory-to-memory language with a semantic content between high-level and conventional assembler languages. An FHP S-language instruction has an 8-bit opcode, followed by some number of operand syllables.

S-language Program Counter
: The FHP architecture uses an S-language Program Counter (PC) to locate a process' currently executing S-instruction opcode. The PC contains a 32-bit offset value.

S-op
: An FHP S-language opcode (see Opcode).

Secure stack
: A non-architectural, per-process stack which holds information that must not be changed by a process. This stack is used to save macro-state during an Architectural Call. The secure stack also stores host machine micro-state when a process is removed from a Virtual Processor. The secure stack is invisible to high-level language programmers.

Short Name Table Entry
: A Name Table Entry that contains a single, 64-bit Name Table element. See Name Table Entry.

Static data
: The class of storage unique to a particular user stack. Static data persists through many activations of a procedure. A new activation of a procedure always inherits the previous activation's static data. Static data is initialized the first time a procedure is activated in a process, and is stored in a data object called the static data area.

Static Data Pointer
(SDP)
: One of the three Architectural Base Registers. The SDP locates data used by a procedure during a previous activation. This data is in the procedure's static data area (see Static data).

Subject
: The unit of authority within the FHP architecture. A subject has three components: a process identifier, a principal identifier, and a domain identifier. Each component of a subject is named by a UID.

Unique Identifier
(UID)
: The 80-bit name of an FHP object. Unique Identifiers (UIDs) are an object's "fingerprint"; they are never reassigned. Once an object is created, it may always be named by its UID. UIDs contain two major fields: a 48-bit Object Serial Number (OSN) and a 32-bit Logical Allocation Unit Identifier (LAUID). FHP systems can use Logical Address Reduction to compress UIDs into more easily manageable Active Object Numbers (see Active Object Number).

User stack
: A data structure which holds information local to a procedure's activation. This information includes automatic data and pointers to any arguments that procedure uses.

User stack frame
: A collection of information on a user stack. A stack frame represents a procedure's activation record.

Vector
: A one-dimensional array.

Virtual Processor
(VP)
: FHP processes are run on virtual rather than real processors. Virtual Processors are created by enhancing a system's host machine with software and firmware.

Working set
: An attribute of a process. While a process is bound to a processor, its working set is the group of memory pages which the system has accelerated to main memory. A working set establishes a limit on the number of page frames a process can have in main memory.

Writable Control Store
(WCS)
: A fast Random Access Memory (RAM) used to store microcode. Most FHP systems will execute S-language interpreters out of Writable Control Store so that the system can switch quickly from one S-interpreter to another.

--End of Appendix--

# Index

Note: Underscored page numbers (e.g., 1-5) indicate definitions of terms or other key information.

7) UN-INITIALIZED BIT ON DATA TYPE
PREVENT A ← B+C when
B+C NOT INITIALIZED

CALL FRED — new frame same stack

+f(x)

+SIN(x) ① run-time NO address
② replace with SIN(x) ⟶ PFD GATE
① coef
② UIT

CALL GTIME — system call

Questions

1) AUN's stored in data base } Roll out
problems of persistence

2) What is justification on NTE

3) Multiple Static / Indirect thru Static Data

4) If dynamic linking
allow execute subroutine
without binding need Name binding
or different names per activation record.

5) MIXED MODE ARITHMETICS
- IF DATA TYPE IN NTE
LEAVE TYPELESS OPERATORS

6) CHECK DATA TYPE ON
SUBROUTINE ARGUMENT CHECKING